2

PROCEEDINGS OF NINTH ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

Sponsored By:
ANCOST, INC.

With Participation By:
UNITED STATES ARMY
UNITED STATES NAVY
UNITED STATES MARINE CORPS
UNITED STATES AIR FORCE
FEDERAL AVIATION ADMINISTRATION
STRATEGIC DEFENSE INITIATIVE OFFICE
ADA JOINT PROGRAM OFFICE
NATIONAL AERONAUTICS & SPACE ADMINISTRATION

Academic Host:
COPPIN STATE COLLEGE

WASHINGTON HILTON & TOWERS - WASHINGTON, DC

March 4-7, 1991

Approved for Public Release: Distribution Unlimited

9th ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY
CONFERENCE COMMITTEE 1990-1991

Executive Committee Chair:
DR. M. SUSAN RICHMAN
The Pennsylvania State
University at Harrisburg
Middletown, PA 17057

Treasurer:
MS. SUSAN MARKEL
TRW
Fairfax, VA 22031

Secretary:
MR. JESSE WILLIAMS
Cheyney University
Cheyney, Pa 19319

Chair-Elect:
MS. DEE M. GRAUMANN
General Dynamics, DSD
San Diego, CA 92123

Immediate Past-Chair:
MR. MICHAEL D. SAPENTER
Telos Federal Systems
Lawton, OK 73501

Conference Co-Chair:
MS. JUDITH M. GILES
Intermetrics
Cambridge, MA 02138

Conference Co-Chair:
MS. CATHERINE PEAVY
Martin Marietta
Information Systems Group
Englewood, CO 80112

Academic Outreach Co-Chair:
MS. LUWANA S. CLEVER
Florida Institute of Tech
Melbourne, FL 32901

Academic Outreach Co-Chair:
MR. JAMES E. WALKER
Network Solutions
Herndon, VA 22070

Budget Committee Chair:
MR. DONALD C. FUHR
Tuskegee University
Tuskegee, AL 36088

By-Laws Committee Chair:
DR. RICHARD KUNTZ
Monmouth College
W Long Branch, NJ 07764

Technical Program Chair:
MS. CHRISTINE L. BRAUN
Contel Technology Center
Fairfax, VA 22033

MR. MIGUEL A. CARRIO, JR.
Teledyne Brown Engineering
Fairfax, VA 22030

MS. VERLYNDA DOBBS
Wright State University
Kettering, OH 45420

MR. DAVID L. JOHNSON
GTE Government Systems
Rockville, MD 20850

DR. ARTHUR JONES
Morehouse College
Atlanta, GA 30314

DR. GENEVIEVE M. KNIGHT
Coppin State College
Baltimore, MD 21216

MR. STEVE LAZEROWICH
Alsys
Reston, VA 22090

DR. CHARLES LILLIE
Science Application Int'l
McLean, VA 22101

MR. RICHARD PEEBLES
Concurrent Computer Corp.
Tinton Falls, NJ 07724

MS. LAURA VEITH
Andrulis Research Corp.
Bethesda, MD 20814

CONFERENCE DIRECTOR:
MARJORIE Y. RISINGER, CMP
Rosenberg & Risinger, Inc.
Culver City, CA 90230

ADVISORY MEMBERS

MR. LOUIS J. BONA
FAA Technical Center
Atlantic City Airport, NJ 08405

MR. DANIEL E. HOCKING
AIRMICS
Atlanta, GA 30332-0800

DR. JOHN SOLOMOND
Ada Joint Program Office
Washington, DC 20310

MR. CARRINGTON STEWART
NASA
Houston, TX 77058

MS. ANTOINETTE STUART
Department of the Navy
Washington, DC 20734

CAPT. DAVID THOMPSON
HQ, U.S. Marine Corps
Washington, DC 20380-0001

MS. KAY TREZZA
HQ, CECOM, CSE
Ft. Monmouth, NJ 07703-5000

MR. GEORGE W. WATTS
HQ, CECOM, CSE
Ft. Monmouth, NJ 07703-5000

PANELS AND TECHNICAL SESSIONS

Tuesday, March 5, 1991

8:30 AM	Opening Session
10:00 AM	Ada Policies, Practices & Initiatives Panel
2:00 PM	Real-Time Issues
2:00 PM	Reuse I
2:00 PM	Management
2:00 PM	Technology Research I
4:00 PM	Reuse II
4:00 PM	Metrics
4:00 PM	Technology Research II

Wednesday, March 6, 1991

8:30 AM	Opening Session
9:00 AM	Ada Success Stories Panel
10:45 AM	Applications
10:45 AM	Education I
10:45 AM	Environments
10:45 AM	Process & Methods I
2:00 PM	Mini Tutorial
2:00 PM	Education II
2:00 PM	Process & Methods II
4:00 PM	Total Quality Management Panel
4:00 PM	Preparing Students for Industry Panel

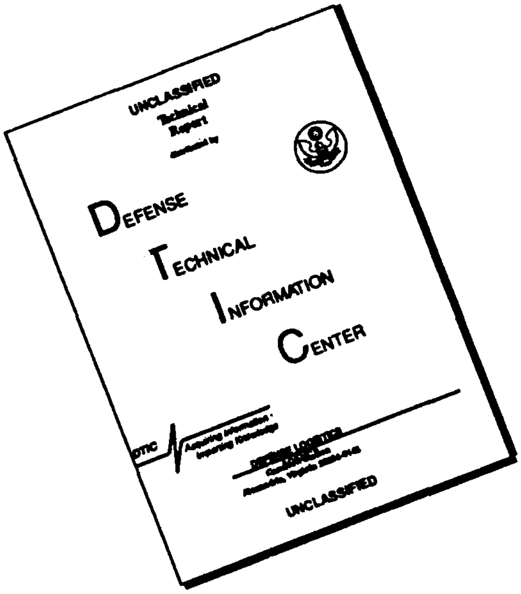
Thursday, March 7, 1991

8:30 AM	SEI Assessment Panel
8:30 AM	Student Papers I
10:30 AM	Ada 9X Panel
10:30 AM	Secure Systems
10:30 AM	Student Papers II
2:00 PM	Future Directions in Ada Panel

Papers

The papers in this volume were printed directly from unedited reproducible copies prepared by the authors. Responsibility for contents rests upon the author, and not the symposium committee or its members. After the symposium, all publication rights of each paper are reserved by their authors, and requests for republication of a paper should be addressed to the appropriate author. Abstracting is permitted, and it would be appreciated if the symposium is credited when abstracts or papers are republished. Requests for individual copies of papers should be addressed to the authors.

DISCLAIMER NOTICE



UNCLASSIFIED
Technical
Report
distributed by

DEFENSE
TECHNICAL
INFORMATION
CENTER

DTIC
Acquiring Information -
Imparting Knowledge

DEFENSE LOGISTICS AGENCY
Cameron Station
Alexandria, Virginia 22304-6145

UNCLASSIFIED

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

PROCEEDINGS

EIGHTH NATIONAL CONFERENCE ON ADA TECHNOLOGIES

Bound—Available at Fort Monmouth

2nd Annual National Conference on Ada Technology Proceedings—1984 (Not Available)
3rd Annual National Conference on Ada Technology Proceedings—1985—$10.00
4th Annual National Conference on Ada Technology Proceedings—1986 (Not Available)
5th Annual National Conference on Ada Technology Proceedings—1987 (Not Available)
6th Annual National Conference on Ada Technology Proceedings—1988—$20.00
7th Annual National Conference on Ada Technology Proceedings—1989—$25.00
8th Annual National Conference on Ada Technology Proceedings—1990—$25.00

Extra Copies: 1-3 $25; next 4 $20; next 11 & above $15 each

Make check or bank draft payable in U.S. Dollars to ANCOST and forward requests to:

Annual National Conference on Ada Technology
U.S. Army Communications-Electronics Command
ATTN: AMSEL-RD-SE-CRM (Ms. Kay Trezza)
Fort Monmouth, New Jersey 07703-5000

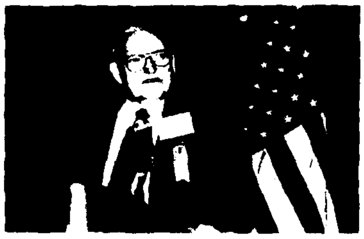
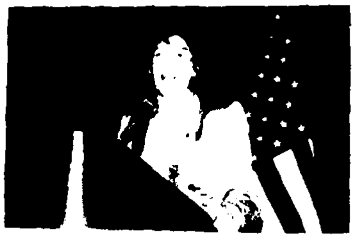
Telephone inquiries may be directed to Ms. Kay Trezza at (201) 532-1898

Photocopies—Available at Department of Commerce. Information on prices and shipping charges should be requested from:

U.S. Department of Commerce
National Technical Information Service
Springfield, Virginia 22151
USA

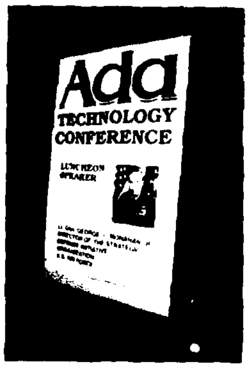
Include title, year, and AD number

2nd Annual National Conference on Ada Technology—1984—AD A142403
3rd Annual National Conference on Ada Technology—1985—AD A164338
4th Annual National Conference on Ada Technology—1986—AD A167802
5th Annual National Conference on Ada Technology—1987—AD A178690
6th Annual National Conference on Ada Technology—1988—AD A190936

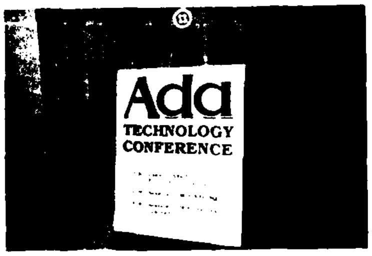



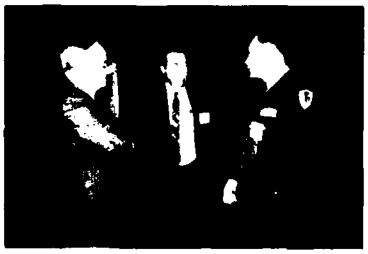
HYATT REGENCY
ATLANTA GA

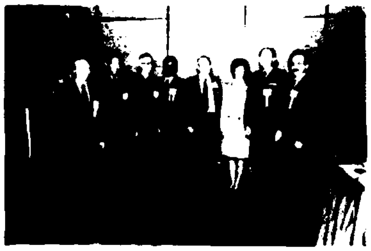
1990 Ada Conference

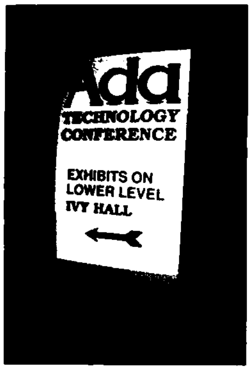


Ada Sessions and Panel Discussions

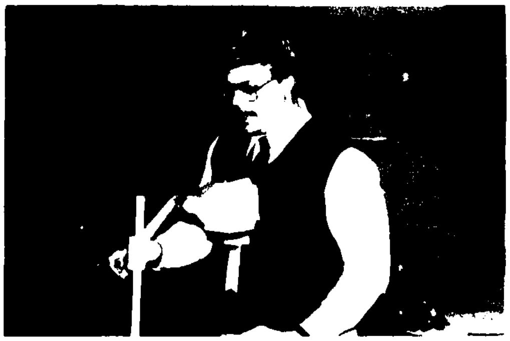




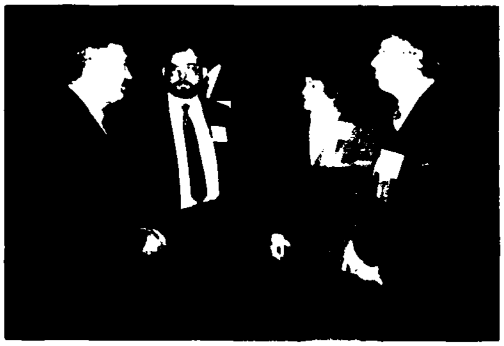




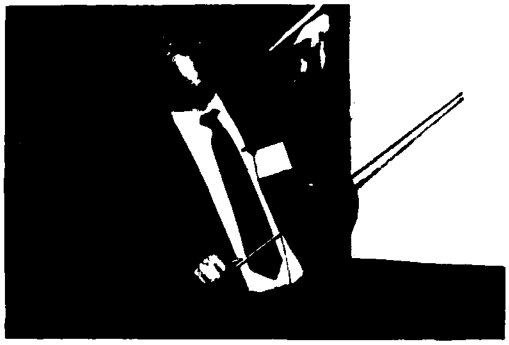
Ada Conference Registration and Exhibits



Ada Conference Highlights



Ada Conference Highlights



Ada Conference Highlights

8th ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

Awards and Thank You's

The 1990 Ada Technology Conference Planning Committee posed for this group picture after a successful conference.

TABLE OF CONTENTS

TUESDAY, MARCH 5, 1991

hibit Hours: 3:30pm – 7:00pm

'ENING SESSION: 8:30am - 9:30am

Keynote Speaker: LTG Billy M. Thomas -- Deputy CommandingGeneral for Research, Development, and Acquisition, HQ Army Materiel Command, Washington, DC 2

ınel Discussion I: 10:00am - 12:00n

la Policies, Practices & Initiatives

Moderator: Mr. John H. Sintic, Director, CECOM Center for Software Engineering, Fort Monmouth, NJ 3

Spotlight Speaker:

VADM Jerry O. Tuttle, U.S. Navy, Director Space & Electronic Warfare, Navy Department, Washington, DC 4

Panelists:

Mrs. Miriam Browning, Vice Director for Information Management for Command, Control, Communication and Computers, U.S. Army, Washington, DC 5

BGEN Gerald L. McKay, Deputy Ass't Chief of Staff for C412 HQ, U.S. Marine Corps, Washington, DC

Mr. James G. Cain, Deputy Associate Administrator for Systems Development, FAA, Washington, DC

Air Force Representative, TBA

JNCH: 12:00n - 2:00pm

Guest Speaker: LTG August M. Cianciolo -- Military Deputy to the Assistant Secretary of the Army, The Pentagon, Washington, DC 6

ession 1: 2:00pm - 3:30pm

eal-Time Issues

Chairperson: Mr. Steve Lazerowich, Alsys, Reston, VA

1 Managing Throughout When Using Ada in a Hard, Real-Time System--W. L. Miller, Georgia Institute of Technology, Atlanta, GA 9

2 Evaluation of Real-Time Distributed Systems Using Ada Concurrency--M. Bassiouni and M. Chiu, University of Central Florida, and J. Thompson, Institute for Simulation and Training, Orlando, FL 16

3 A Dynamic Preference Control Solution to the Readers and Writers Problem--S. Miriyala, University of Illinois at Urbana-Champaign, Urbana, IL, and T. Elrad, Illinois Institute of Technology, Chicago, IL 25

ession 2: 2:00pm - 3:30pm

euse I

Chairperson: Ms. Christine L. Braun, Contel Technology Center, Chantilly, VA

1 The RAPID Center Reusable Software Components (RSCs) Certification Process--J.C. Piper and W. L. Barner, U.S. Army Information Systems Software Development Center-Washington, Fort Belvoir, VA 32

2 STARS Asset Management System--B.J. Kitaoka and P.A. Berggren, Science Applications International Corp, Orlando, FL

3 Software Reuse Progress--J.W. Hooper, The University of Alabama in Huntsville, Huntsville, AL and R.O. Chester, Martin Marietta Energy Systems, Inc., Oak Ridge, TN 39

ession 3: 2:00pm - 3:30pm

lanagement

Chairperson: Ms. Judith Giles, Intermetrics, Inc., Cambridge, MA

.1 An Integrated Approach to Ada Project Planning--B. Papanicolaou, Lockheed Sanders, Nashua, NH 45

.2 Managing the Ada Conversion and Integration of Mission Critical Defense Systems--T.S. Archer, Contel Telos Corp., Sierra Vista, AZ 50

3.3 On Decomposing an Ada CSCI of a Large Command and Control System into TLCSCs, LLCSCs and Units: With Suggestions for Using DOD-STD-2167A--L. Gray, Ada PROS, Inc., Fairfax, VA. .. 55

Session 4: 2:00pm - 3:30pm

Technology Research I

Chairperson: Dr. Charles Lillie, Science Application International, McLean, VA

4.1 Automatic Programming Systems for Ada: Theory and Practice of Object Oriented Methods of Program Specification from Requirements-W. Arden, Contel Telos Corp, Shrewsbury, NJ 69

4.2 An Approach to Ada Implementation of an Associative Memory System--E.K. Park and F.A Skove, U.S. Naval Academy, Annapolis, MD, and C.S. Kang, The American University, Washington, DC 72

4.3 Incorporating Garbage Collection and Persistence within Ada--S.J. Zeil, Old Dominion University, Norfolk, VA 80

Session 6: 4:00pm - 5:30pm

Reuse II

Chairperson: Mr. Daniel Hocking, AIRMICS, Atlanta, GA

6.1 Analysis of Software Reuse on AFATDS Concept Evaluation--C.A. Burnham, R.D. Gerardi, P. Ho, and H.F Joiner, II, Contel Telos Corp., Shrewsbury, NJ 87

6.2 A Methodology for the Evaluation of Reusable Ada Software Libraries--S.H. Levy, Calculemics, Inc., Fanwood, NJ 92

6.3 Facilitating Reuse in a Software Engineering Environment--J.C. Schettino, Jr. and C. Kozlowski, Contel Technology Center, Chantilly, VA 100

Session 7: 4:00pm - 5:30pm

Metrics

Chairperson: Mr. Michael Sapenter, Contel Telos Corp., Lawton, OK

7.1 Measuring Ada Design to Predict Maintainability--W. Li, S. Henry, and C. Selig, Virginia Polytechnic Institute, Blacksburg, VA 107

7.2 A Software Metrics Database: Support for Analysis and Decision-Making--S.L. Pfleeger and J.C. Fitzgerald, Jr., Contel Technology Center, Chantilly, VA 114

7.3 A Distributed Compilation Environment-Lessons Learned--D. Gotterbarn, East Tennessee State University, Johnson City, TN 120

Session 8: 4:00pm - 5:30pm

Technology Research II

Chairperson: Dr. Arthur Jones, Morehouse College, Atlanta, GA

8.1 Incremental Operational Specifications for the Verification of Ada Programs--B. Wieand and W. Howden, University of California-San Diego, La Jolla, CA 128

8.2 N-Version Programming Using the Ada Tasking Model--R.J. Leach and D.M. Coleman, Howard Univ., Washington, DC 135

8.3 Concurrency in Ada and VHDL--C. Schaefer, The MITRE Corp., McLean, VA. 142

5:30pm - 7:00pm -- Reception in Exhibit Hall

7:00pm - 10:00pm -- "Birds of a Feather" Discussions

WEDNESDAY, MARCH 6, 1991

Exhibit Hours: 10:30am - 7:00pm

KEYNOTE SPEAKER: 8:30am - 9:00am

LTG Jerome B. Hilmes -- Director, Directorate of Information Systems for Command, Control, Communication and Computers, Office of the Secretary of the Army, The Pentagon, Washington, DC 7

Panel Discussion II: 9:00am - 10:30am
Ada Success Stories
Moderator: Ms. Dee M. Graumann, General Dynamics, DSD, San Diego, CA
Panelists:
Mr. Stan Levine, Deputy Project Manager, Common Hardware Software, Ft. Monmouth, NJ
Capt. Samuel Jordan, Head User View Systems Branch, U.S. Marine Corps, Central Design & Programming Activity
Mr. Mike Stark, Goddard Space Flight Center, Greenbelt, MD
Mr. Currie Colket, Navy Deputy to the AJPO, The Pentagon, Washington, DC
Mr. Thomas P. O'Mahoney, Program Director, Air Base Decision System, Hanscom AFB, MA

Session 9: 10:45am - 12:15pm
Applications
Chairperson: Mr. George Watts, CECOM CSE, Fort Monmouth, NJ
9.1 Developing Interface Standards for Ada Systems--E.L. Krous, Contel Telos Corp., Lawton, OK 150
9.2 A Layered Architecture for DBMS Interactions from an Ada Application--A. McKay and R.A. Pederson, GTE Government Systems, Rockville, MD 152

Session 10: 10:45am - 12:15pm
Education I
Chairperson: Ms. Luwana Clever, Florida Institute of Technology, Melbourne, FL
10.1 Tasking Facilities in Ada: "The Cigarette Smokers Problem"--T.L. Newsome, Lanham, MD 160
10.2 Adopting Ada as a Primary Undergraduate Programming Language --J.J. Spegele and E.K. Park, U.S. Naval Academy, Annapolis, MD 164
10.3 Classroom Activities for an Advanced Ada Class--F.L. Moore, Texas Instruments, Inc., Dallas, TX 170

Session 11: 10:45am - 12:15pm
Environments
Chairperson: Mr. Carrington Stewart, NASA, Lyndon B.Johnson Space Center, Houston, TX
11.1 Architecting Distributed Real-Time Ada Applications: The Software Architect's Life Cycle Environment--W.E. Royce and D.L. Brown, TRW SEDD, Redondo Beach, CA 174
11.2 Universal Network Architecture Services: A Portability Case Study--W.E Royce, P.W. Blankenship, E.A. Rusis, and B.P. Willis, TRW SEDD, Redondo Beach, CA 181
11.3 The Catalyst Software Engineering Environment--S.L. Mulholland, Rockwell International, Cedar Rapids, IA 188

Session 12: 10:45am - 12:15pm
Process and Methods I
Chairperson: Ms. Catherine Peavy, Martin Marietta Information Systems Group, Englewood, CO
12.1 Transforming the 2167A Requirements Definition Model into an Ada-Object Oriented Design--J.T. Lukman, Thousand Oaks, CA 200
12.2 An Overview of the Clear Lake Life Cycle Model (CLLCM)-- K.L. Rogers, The MITRE Corp., Houston, TX; M. Bishop, Unisys, Houston, TX; and C.W. McKay, NASA Software Engineering Research Center, University of Houston-Clear Lake, Houston, TX 206
12.3 Quality Assurance Requirements for an Evolutionary Development Methodology--R.M. Lobsitz, P.G. Clark, and C.R. String, TASC, Reading, MA 223

LUNCH BREAK: 12:15pm - 2:00pm

Mini Tutorial: 2:00pm - 3:30pm
Object Oriented Software Engineering
Moderator: Mr. Edward V. Berard, President, Berard Software Engineering Inc., Germantown, MD
This mini tutorial will discuss Object Oriented Software Engineering (OOSE) and include these topics:
•Object oriented requirements analysis (OORA)
•Object oriented design (OOD)
•Object oriented domain analysis (OODA)

Ada specific issues will be mentioned where appropriate by the presenter, who has over 8 years' experience in applying object oriented technology.

Session 13: 2:00pm - 3:30pm
Education II
Chairperson: Dr. Genevieve M. Knight, Coppin State College, Baltimore, MD
13.1 Software Engineering Education "An Experiment"--P.B. Lawhead, The University of Mississippi, University, MS, and R. Hess, IBM Corporation, Boulder, CO 233
13.2 Leveraging CBT in Universities to Produce Productive Ada Students for Industry--J.E. Walker, Network Solutions, Herndon, VA 238
13.3 An Ada-Based Translator Writer System--T.F. Reid, Contel Technology Center, Chantilly, VA 243

Session 15: 2:00pm - 3:30pm
Process and Methods II
Chairperson: Ms. Susan Markel, TRW, Fairfax, VA
15.1 *Quality Assurance Using Ada and DOD-STD-2167A*-- D.L. Disbrow and J.D. Martin, Telos Systems Group, Lawton, OK 249
15.2 A Look at SEI Software Process Assessments--T.B. Bollinger and C. McGowan, Contel Technology Ctr, Chantilly, VA 255

Panel Discussion IV: 4:00pm - 5:30pm
Total Quality Management
Moderator: Dr. Alan B. Salisbury, President, Contel Technology Center, Chantilly, VA
Panelists:
Dr. David Sundstrom, General Dynamics, Fort Worth, Tx
Mr. Dennis Turner, CECOM CSE, Ft. Monmouth, NJ
Mr. David Owens, Software Productivity Consortium, Herndon, VA

Panel Discussion V: 4:00pm - 5:30pm
Preparing Students for Industry
Moderator: Mr. James Walker, Network Solutions, Herndon, VA
Panelists:
Ms. Pamela Lawhead, University of Mississippi, University, MS.
Mr. Johan Margano, Computer Science Corp., Rockville, MD

5:30pm - 7:00pm – Reception in Exhibit Area

THURSDAY, MARCH 7, 1991
Exhibit Hours: 8:00am - 12:00n

Panel Discussion VI: 8:30am - 10:00am
SEI Assessment
Moderator: Ms. Susan Markel, TRW, Fairfax, VA
Panelists:
Mr. Kenneth Dymond, SEI, Pittsburgh, PA
Mr. Robert Beach, TRW, Redondo Beach, CA
Mr. Judah Mojalinski, Contel Technology Center, Chantilly, VA

Student Papers I: 8:30am - 10:00am

Panel Discussion VII: 10:30am - 12:00n
Ada 9X
Moderator: Ms. Christine Anderson, U.S. Dept. of Defense, Elgin AFB, FL
Panelists: Key Participants in the Ada 9X Process

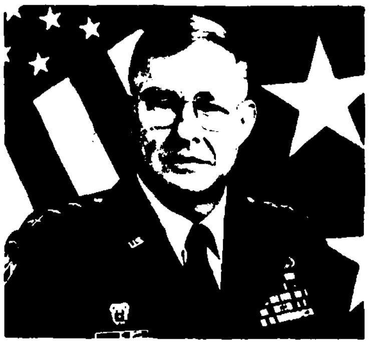
Session 16: 10:30am - 12:00n
Secure Systems
Chairperson: Mr. Donald C. Fuhr, Tuskegee University, Tuskegee, AL
16.1 The Mystery: Why do Many of the Variables Declared in Ada Programs Model Conceptually Invariant Objects?–J.A. Perkins, Dynamics Research Corp., Andover, MA 282
16.2 Programming Practices Relating to Minimal Visibility and Minimal Interaction--J.A. Perkins, Dynamics Research Corp., Andover, MA 294

Student Papers II: 10:30am - 12:00n

LUNCH – 12:00n - 2:00pm
Guest Speaker: MG Albert J. Edmonds -- Assistant Deputy Chief of Staff, Command, Control, Communication and Computers, HQ, USAF, Washington, DC 8

Panel Discussion VIII: 2:00pm - 4:00pm
Future Directions in Ada
Moderator: Dr. Barry Boehm, DARPA, Washington DC
Panelists:
Mr. Marshall Potter, U.S. Navy
FAA Representative

KEYNOTE SPEAKER



LTG Billy M. Thomas
Deputy Commanding General for Research,
Development and Acquisitions,
HQ, U.S. Army Materiel Command
Alexandria, VA

Lieutenant General Thomas was born in Crystal City, Texas and grew up in Kileen. His college years were spent at Texas Christian University where, in 1962, he completed the Reserve Officer's Training Corps curriculum, earned a BS Degree in Secondary Education, and was then commissioned a second lieutenant in the U.S. Army. General Thomas also holds an MS Degree in Telecommunications Operations from George Washington University.

General Thomas has attended the Signal School, Basic and Advanced Courses, the U.S. Army Command and General Staff College, and the U.S. Army War College.

In over 27 years' of active service, he has had important overseas command assignments in Germany, Thailand, and Vietnam, and held a variety of significant staff assignments prior to his present position. His most recent include Commanding General,

Communications-Electronics Command and Fort Monmouth, Deputy Director, Combat Support Systems, Office of the Deputy Chief of Staff for Research, Development and Acquisition, Washington, DC, and Deputy Commander/Assistant Commandant, U.S. Army Signal Center and School, Fort Gordon, GA.

Decorations and Badges awarded to General Thomas include the Legion of Merit with Oak Leaf Cluster, Bronze Star with 2 Oak Leaf Clusters, Meritorious Service Medal with 3 Oak Leaf Clusters, and the Joint Service Commendation Medal. He is also authorized to wear the Parachutist's Badge.

He is married to the former Judith K. McConnell of Boise, Idaho.
They have four children.

OPENING PANEL

Ada POLICIES, PRACTICES, AND INITIATIVES

Mr. John H. Sintic
Director, CECOM Center for
Software Engineering, U.S. Army CECOM
Fort Monmouth, NJ

John H. Sintic assumed his current position as Director, CECOM Center for Software Engineering (CECOM CSE), on 1 April 1988. The CECOM Center is the single CECOM focal point for providing software life cycle management, software engineering and software support to Mission Critical Defense Systems (MCDSs) used in strategic and tactical Battlefield Functional Areas (BFAs) supported by CECOM. The CECOM CSE is also the Army/Army Material Command focal point for Computer Resource Management (CRM), Advanced Software Technology (AST), Ada Technology, Joint/Army Interoperability Testing (JAIT), and software quality and productivity.

Mr. Sintic has been with the Center since December, 1983. Prior to his present assignment, he served as Deputy Director of the CECOM CSE. He also served as Associate Director, Computer Resource Management and Software Engineering Support, CECOM CSE.
Mr. Sintic has over 25 years of experience in the field of software and computer technology.

Before joining the Center, Mr. Sintic was Chief of the Engineering Division, Joint Interface Test Force/Joint Interoperability Tactical Command and Control Systems (JITF/ JINTACCS) from 1978 to 1983. In this position, he directed engineers and computer scientists (military and civilian) in the research, development and engineering for Joint Service Interoperability of Command, Control and Communications Systems.
He served as project manager for the development of the Joint Interface Test Systems (JITS) - the world's largest distributed command and control interoperability test bed developed to eleven Joint Service/Agency test sites.

Mr. Sintic has a BS Degree in computer science. He is involved in many civic functions and has served on the Ocean Township Board of Education. He is also a member of the Monmouth College High Technology Advisory Board.

John and Trudy Sintic have four sons and live in Oakhurst, NJ.

SPOTLIGHT SPEAKER
OPENING PANEL

Vice Admiral Jerry O. Tuttle, U.S. Navy
Director, Space and Electronic Warfare
Office of the Chief of Naval Operations

Before assuming his current assignment, Vice Admiral Tuttle's career has included assignments as Deputy Director for Intelligence and External Affairs at the Defense Intelligence Agency; Deputy and Chief of Staff for the Commander in Chief, U.S. Atlantic Fleet; and Director, Command, Control and Communications System, the Joint Chiefs of Staff. He has commanded Attack Squadron EIGHT ONE; Carrier Air Wing THREE; replenishment ship USS KALAMAZOO; Aircraft Carrier USS JOHN F. KENNEDY, Carrier Group EIGHT; and Carrier Group TWO/Battle Force SIXTH Fleet.

Vice Admiral Tuttle received a Communications Engineering degree from the Naval Postgraduate School in 1963 having attended the undergraduate and postgraduate schools simultaneously. He graduated with honors from the Naval War College, Newport, Rhode Island, and concurrently received an MA in International Relations from George Washington University in 1969.

Vice Admiral Tuttle's personal decorations include the Defense Distinguished Service Medal; Distinguished Service Medal; Defense Superior Service Medal; Legion of Merit (4);Distinguished Flying Cross (3); Meritorious Service Medal (2); Air Medal (23) - five individual and 18 strike/flight awards; Navy Commendation Medal (4); and various campaign awards. He flew over 200 combat missions over North Vietnam and has more than 1000 arrested carrier landings.

Vice Admiral Tuttle is married to the former Barbara Bonifay of Pensacola, FL. They have five children, Michael, Vicky, Mark, Stephen and Monique.

OPENING PANEL
Ada POLICY, PRACTICES, AND INTIATIVES

Miriam F. Browning
Vice Director for Information Management Office
Director of Information Systems
for Command, Control, Communications and Computers,ODISC4

Miriam F. Browning serves as the senior civilian for information resources management for the Department of the Army.

Prior to this position, Mrs. Browning was the Deputy Assistant Inspector General for Administration and Information Management, DoD, Office of the Inspector General.

Mrs. Browning has held a variety of important positions in government. She has been in Senior Management positions with Deputy Chief of Staff for Resource Management and Deputy Chief of Staff for Information Management at HQ's Forces Command. She was also the Chief of Computer Services for the Center for Infectious Diseases for the Centers for Disease Control.

Mrs. Browning has a BA from Ohio State University in Political Science, and an MS in Information Technology from George Washington University. Her military education includes Federal Executive Institute in Charlottesville, VA and US Army War College, Carlisle Barracks, PA.

Mrs. Browning has received numerous awards including the DA Meritorious Civilian Service Award in 1988.

She and her husband, David Benjamin Browning of Edenton, NC, reside in Alexandria, VA.

LUNCHEON SPEAKER

Lieutenant General August M. Cianciolo
Military Deputy to the Assistant Secretary of the Army
(Research, Development and Acquisition)

Lieutenant General August M. Cianciolo, as Military Deputy to the Assistant Secretary of the Army for Research, Development and Acquisition, supports the Army Acquisition Executive and the Assistant Secretary of the Army for RDA with staff work, advice, and decision recommendations for the breadth of the Army Acquisition function; places major emphasis on Technology and Assessment, Systems, and Plans and Programs; serves as chairman of the Preliminary Army Systems Acquisition Review Committee (ASARC); is the principal military witness for RDA appropriations with the Congress; advises the Army Chief of Staff and Vice Chief of Staff on research, development and acquisition; supervises the Program Executive Officer system.

He is the former Deputy Commanding General for Research, Development and Acquisition at the Army Material Command. Prior to this position he was Commanding General at the U.S. Army Missile Command at Redstone Arsenal.

Lieutenant General Cianciolo has held a variety of important positions including Deputy for Systems Management in the Office of the Assistant Secretary of the Army for Research, Development and Acquisition. He was the Deputy Director of Material, Plans and Programs and later the Deputy Chief of Staff for Research, Development and Acquisition. Lieutenant General Cianciolo was the Project Manager for Multiple Launch Rocket System at MICOM as well as the Project Manager for the Standoff Target Acquisition/Attack System at Fort Monmouth.

Lieutenant General Cianciolo received a BA in Accounting from Xavier University and MS Degree in Aerospace Engineering at the University of Southern California. His military education includes the Field Artillery School Basic Course, the Air Defense Artillery School Advanced Course, the United States Army Command and General Staff College and the United States Army War College.

Awards and decorations which Lieutenant General Cianciolo has received include the Distinguished Service Medal, the Bronze Star Medal with "V" Device and two Oak Leaf Clusters, the Meritorious Service Medal with one Oak Leaf Cluster, Air Medals, the Army Commendation Medal with two Oak Leaf Clusters and the Master Army Aviator Badge.

He and his wife, Sheila, have one daughter, Theresa and two sons, Martin and Anthony.

KEYNOTE SPEAKER

LTG Jerome B. Hilmes
Director of Information Systems for
Command, Control, Communications, and
Computers, U.S. Army

Lieutenant General Jerome B. Hilmes was born in Carlyle, Illinois on 21 December 1935. Upon completion of studies at the United States Military Academy in 1959, he was commissioned a second lieutenant and awarded a Bachelor of Science degree. He also holds a MS and PhD degree from Iowa State University. His military education includes completion of the U.S. Army Engineer School, U.S. Army Command and General Staff College and the Naval War College. He is a registered professional engineer in the state of New York.

He has held a wide variety of command and staff positions culminating in his current assignment as Director of Information Systems for C4, Office Secretary of the Army. These include:

Commander of OTEA, a field operating agency of the Office of the Chief of Staff, Army. General Hilmes had responsibility for planning and conducting continuous comprehensive evaluation (C2E) and operational testing for all major Army systems.

Commander of the Southwestern Division, U.S. Army Corps of Engineers in Dallas (1985-88) and the Corps' North Central division in Chicago (1983-85). He was also Chairman of the Board of Engineers and a member of the Mississippi River Commission.

Commander, 7th Engineer Brigade and Ludwigsburg-Kornwestheim Military Community, Stuttgart, Germany (1978-80); Commander, 23rd Engineering Battalion, 3rd Armored Division, Hanau, Germany (1976); and Commander, Task Force Sierra, 18th Engineer Brigade, Vietnam (1970-71).

General Hilmes' major staff assignments include Deputy Assistant Chief of Engineers and Programs, Washington, DC (1981-83); Director of Facilities Engineering and Housing, Fort Bragg, NC (1980-81); and Assistant Deputy Chief of Staff Engineer Headquarters, U.S. Army, Europe (1976-78).

His publications include, "LHX", "Green Ribbon Panel Report, Mar 85", "Winter Reforger '79: Lessons Learned", "Stopping an Armored Attack", MICV-UTTAS", "Statistical Analysis of Under-reinforced Prestressed concrete Flexural Members" and "Seventh Army Expedient Bridge".

Awards and decorations which General Hilmes received include the Distinguished Service Medal, Legion of Merit (three awards), Bronze Star Medal (two awards), Meritorious Service Medal, Air Medal, Joint Service Commendation Medal, and Army Commendation Medal with V Device (two awards) and the Gallantry Cross with Silver Star.

He is married to the former Geri McDonough of Albany, NY. They have four sons: Bruce, Gary, Douglas and Andrew. Bruce and Gary are both in the U.S. Army.

LUNCHEON SPEAKER

MAJGEN Albert J. Edmonds
Assistant Deputy Chief of Staff
Command, Control, Communications & Computers, HQ, USAF

Major General Albert J. Edmonds is Assistant Chief of Staff/Systems for Command, Control, Communications and Computers, Headquarters United States Air Force, Washington, DC. He is responsible for establishing policy for communications and computer systems throughout the Air Force.

General Edmonds received a Bachelor of Science degree in chemistry for Morris Brown College and a Master of Arts degree in counseling psychology from Hampton Institute. He entered the Air Force in 1964 and was commissioned upon graduation from Officer Training School, Lackland Air Force Base, Texas and graduated from Air War College as a distinguished graduate. He completed the National Security program for senior officials at Harvard University in 1987.

The General was assigned to Air Force Headquarters in May 1973. As an Action Officer in the Directorate of Command, Control and Communications, he was responsible for managing Air Communications programs in the Continental United States, Alaska, Canada,, South America, Greenland, and Iceland. In June of 1975 the General was assigned to the Defense Communications Agency and headed the Commercial Communications Policy Office. General Edmonds was assigned to Andersen Air Force Base, Guam, in 1977, as Director of Communications-Electronics for Strategic Air Command's 3rd Air Division and as Commander of the 27th Communications Squadron,

After completing Air War College in June 1980, he returned to Air Force headquarters as Chief of the Joint Matters Group, Directorate of Command, Control and Telecommunications, office of the Deputy Chief of Staff, Plans and Operations. From June 1, 1983 to June 14, 1983 he served as Director of Plans and Programs for the Assistant Chief of Staff for Information Systems.

General Edmonds then was assigned to headquarters Tactical Air Commmand Langley Air Force Base, as Assistant Deputy Chief of Staff for Communications and Electronics, and Vice Commander, Tactical Communications Division. In January 1985 he became Deputy Chief of Staff for Communications-Computer Systems, Tactical Air Command Headquarters, and Commander, Tactical Communications Division, Air Force Communications Command, Langley. In July 1988 he became Director of Command and Control,, Communications and Computer Systems Directorate, U.S. Central Command, MacDill Air Force Base, FL. He assumed his present duties in May 1989.

His military decorations and awards include the Defense Distinguished Service Medal, Legion of Merit Meritorious Service Medal with two oak leaf clusters, and Air Force Commendation Medal with three oak leaf clusters .

The General was named in Outstanding Young Men of America in 1973. he is a member of Kappa Delta Pi Honor Society and is a life member of the Armed Forces Communications and Electronics Association.

General Edmonds is married to the former Jacquelyn Y. McDaniel of Biloxi, MS. They have three daughters: Gia, Sheri and Alicia.

MANAGING THROUGHPUT WHEN USING ADA IN A HARD REAL TIME SYSTEM

Bill Miller

Engineering Sciences Laboratory
Georgia Tech Research Institute

ABSTRACT

Anyone who undertakes the effort to develop Ada software (or any software for that matter) that has the potential for overutilizing the available capacity of a computer, must have a structured and organized method for managing the throughput. This management process consists not only of estimating and tracking the currently used throughput, but also of identifying the potential trouble spots and developing alternatives in advance of their actual need. Any throughput management process that is developed must be easy to use and provide visibility during all stages of the development process. This paper presents a method that can be used by a development team throughout the development cycle of an Ada project.

Most complex systems containing software that are developed for the DOD require use of the Ada programming language. On many systems this doesn't prove to be much of a problem. The code that is being produced by current Ada compilers and the efficiency of run time systems keep improving. As a result there is no reason not to use Ada. However, even with the current (and perhaps future) efficient Ada compilers, hard real-time systems such as signal processing applications or missile guidance applications still provide such a significant timing challenge to software engineers, that a 100% Ada application may not be possible.

The solution is not to throw out the Ada requirement. Aside from it's life cycle cost benefits, Ada as a design tool and programming language simply has too much to offer. And frankly, the increasing number of Ada successes is going to make it much harder to dodge any Ada requirement.

What then is the solution for software contractors who are faced with the requirement to use Ada in a hard real-time application? Or how do you, the organization who is paying for real-time software, convince yourself that your contractor has the throughput situation under control? How does one meet the Ada requirement and still create software that executes within the processor's available throughput?

The typical software manager and designer are faced with the following problems/constraints in hard real-time systems:

- Use the Ada language.

- Usable throughput which could already be insufficient, may be restricted to some percentage of that which is available (typically 60%).

- System/software requirements, although somewhat firm, will continue to "float" until shortly before the product is to be field tested. This adds to the level of throughput uncertainty.

What then is an approach to getting control of this problem? At an early stage in a project's life, it is important to develop an estimate of the amount of throughput that will be used by the software. This estimate is useful not only from the point of satisfying program managers and customers, but also gives you, the software lead or manager, an indication of what type of challenges lay ahead.

Because this estimate quickly becomes an important number that must be defended and/or sold, it is important to develop the best estimate possible with the small amount of information that is available. If the estimate of throughput is overly optimistic, it may make customers and program managers happy for a while. But this optimistic estimate could result in undersizing a processor or processors and could result in expensive hardware redesigns at later stages of the project. An overly pessimistic estimate could result in selecting processors that are too powerful for the application at hand, resulting in an increased and unnecessary hardware life-cycle cost or perhaps contribute to a project cancellation.

This estimate should initially be realistic and somewhat conservative but it should also be considered as a "living" estimate that will be updated and reviewed continually throughout the project until the estimate becomes a set of actual measurements.

Within real time systems there are usually two types of time constraints that are imposed: the first time constraint deals with those tasks that must be performed within a certain period of time (for example: all interrupts must be serviced, statused, logged and reset within 30 microseconds). The second time constraint is that the total software package, outer loop or whatever it's called, execute within a certain period of time (for example: one loop of the software under a worst case situation should take no more than 60 milliseconds and there should be 40 spare milliseconds until the start of the next cycle).

Determining if the processor can appropriately respond to interrupts is fairly easy. Determining if the second requirement can be met is not as easy. A critical factor in developing any overall throughput estimate for a real time system is deciding if floating point operations will be used. One method for estimating throughput involves estimating instruction mixes and total number of lines of code and then looking at compiler vendors' PWIG times or developing your own time measurements and calculating a throughput number. Attempting to do this for the complete software package is a quick way to generate a number. That's about the best thing that can be said for the method, its quick ... not very accurate, but quick.

A better way is to use a preliminary software design that breaks the software into functional blocks. Then use the above, or similar method to estimate a time for each of the blocks. Based on the individual functional timings, develop an aggregate time. An estimate based on a large number of small functional blocks will generate a more accurate estimate than one based on a small number of large functional blocks. It will, however, require more effort to generate this estimate. The following specific method has been used successfully by the author to develop throughput estimates:

IA is the average number of instructions per line of Ada code.

SA is the average instruction speed, including wait states for the processor.

M_x is Any given module

R_x is the number of times per second that the module is executed.

L_x is the estimated or measured number of lines of Ada for the module.

I_x is the calculated number of machine instructions for a given module.

SLT_x is the calculated or measured time required for the module to execute a single loop.

TMT_x is the calculated time slice that a module will use every second.

The following calculation equations are used in building up the throughput estimate for each module:

$$I_x = L_x * IA$$

$$SLT_x = I_x * SA$$

$$TMT_x = SLT_x * R_x$$

Then it is appropriate to combine all of the module throughput estimates into an aggregate number:

$$Total\ time = \sum_{x=1}^{N} TMT_x$$

Figure 1 is a table that should assist in not only developing the estimate but also in maintaining this estimate. Each of the estimates, measurements and calculations described previously are incorporated in this table. As software development progresses, this table will transition from a table of estimates to a table of actual, precise measurements. The appropriate time to update this table would be following the completion of any activity for any module:

- following detailed design completion
- following completion of coding
- following unit test
- following software integration

Once an initial estimate of throughput is developed, it is necessary to develop a throughput budget and to also develop a method for measuring and reporting status against the established budget. When developing a budget it is extremely important to organize the budget in a way that reviewers (who may have little, if any software expertise) can understand the throughput situation and comprehend the impacts of software-related decisions that may be driven by the relationship of the estimate/measurement to the budget. A key factor in developing a method for reporting throughput is to distill the status, removing the "noise-level" software issues and presenting a salient, succinct summary of the situation.

For each module two budgets are developed: a 60% budget and a 95% budget. These budgets, which are only developed once, are based upon the initial throughput estimate and are essentially a proportional scaling of the Time Slice, TMT_x, for each module and the value for Total Time.

M_x is Any given module

$T60B_x$ is the 60% budget.

$T95B_x$ is the 95% budget.

Module Name	Execution Rate Times/Second	Lines of Ada	Number of Machine Instructions	Single Loop Time	1 Second Time Slice
Total Throughput / 1 Second					
Average Machine Instructions / Line of Ada: _________			Average Instruction Speed: ___________		

Figure 1 - Throughput Development Chart

The following calculations are used to develop values for the throughput budgets for each module:

$$T60B_x = (TMT_x \ / \ \text{Total Time}) * .6$$

$$T95B_x = (TMT_x \ / \ \text{Total Time}) * .95$$

These two throughput budgets, the 60% budget and the 95% budget are used to determine the used and available throughput as follows: the 95% BUDGET, which, if met by each module would result in a 100% processor duty cycle. Meeting the 60% BUDGET would result in the not only the processor having sufficient throughput, but it would allow for growth.

Figure 2 presents a method for communicating throughput status that is a modification of the "Processing Time Table" in DI-MCCR-80012A. Each column in Figure exists to aid in conveying the level of confidence and the potential for reduction in throughput.

- The Development State indicates whether the software has completed design, code, test or integration. If development has not started, then some consistent term to describe this should be used.

- The Language indicates if the throughput estimate for the module is for a purely Ada module, a mixture of Ada and assembly or pure assembly.

- The Estimate/Measurement is the throughput number taken from the 1 Second Time Slice column of the table in Figure 1.

- The 60% and 95% Budgets have been describe above.

- The Status column is a Red, Yellow or Green indicator:

    - Green indicates that the 60% budget is being met.

    - Yellow indicates that the estimate/measurement is between the 60% and 95% budgets.

    - Red indicates that a module's throughput exceeds the 95% budget.

Using a chart such as this can give a quick and concise view of the software's throughput situation relative to a given processor a processor clock rate.

Module Name	Development State	Language	Estimate/ Measurement	60% Budget	95% Budget	Status
Total Throughput / 1 Second						
Status Key:	Green Throuput < 60%		Yellow 60% < Throughput < 95%		Red Througput > 95%	

Figure 2 - Throughput Status Chart

At this stage not only has an initial estimate been developed, but also a budget exists. As a next step, a sensitivity analysis of the estimate should be performed to determine if the current processor can support the indicated throughput load or if the processor has not yet been selected, what processors could support the indicated throughput load. Additionally, it should be determined if there are (would be) sufficient processor throughput margins to allow for growth during the project.

Growth should be expected and anticipated because of additions to requirements, compiler uncertainties and the like. Experience indicates that if the first estimate shows that the software would use more than 50% of a processor's available throughput, then a problem exists a there must be either a resizing of the hardware, the software requirements or both. As development for all software modules begins, each module should initially be coded, tested and timed in Ada. Following the initial timings of a couple of modules, it will become clear as to whether or not the throughput budgets are realistic. If all/most of the modules grossly exceed their budgets then, once again, a major problem exists and the processor is probably undersized. If, however, the timings fall into the yellow or green area, with a couple of modules in the red area, then the processor is most likely properly sized.

Not only is it important to develop an estimate of throughput, it is also very important to track and update this estimate as software is developed. During the software development process and the associated throughput tracking process, as various modules transition from Green to Red, bottlenecks will be identified. As potential throughput bottlenecks are identified, workaround alternatives can be developed. If an actual bottleneck is encountered, it should not come as a complete surprise and the previously developed alternatives can be evaluated and implemented.

The next step is attempting to balance the Ada requirement with the available processing power. As was mentioned above, the original development strategy should be to develop all modules in Ada. This will accomplish at least two objectives: by coding all modules in Ada, the spirit of the Ada requirement is met. Secondly, if it should be necessary to convert some modules to assembly language, there will remain a "functioning" pseudo-code module that could act as an ultimate piece of documentation for the module.

The methodology described above deals with predicting throughput, measuring throughput and identifying current or potential throughput problems. Once a problem is identified what do you do? What you don't do is immediately start converting to assembly language every module whose timing does not meet its budget.

When evaluating whether or not to develop an alternative implementation for a particular module, give careful consideration to those modules that have a high R_x, the execution rate. Any throughput savings that is accomplished, will have a multiplying factor. For example if you have two modules, one that executes once per second and one that executes at a 100 Hz. rate, assume that you can cut 1 microsecond from each module. The net saving in the first module is only 1 microsecond. However, the net saving in the second case is 100 microseconds.

In nearly all development efforts for real time systems, the time eventually arrives when it becomes necessary to improve the timing of selected modules. There are a number of ways to improve the execution time of a given module. The alternatives most frequently evaluated are listed below (there is no order of preference implied):

1. Remove generics.
2. Remove procedure calls.
3. Convert floating point calculations to integer or fixed point calculations.
4. Restructure / Redesign the algorithm.
5. Increase the clock speed of the processor or reduce the memory wait states.
6. Convert to assembly.

The option of removing generics generally offers only minimal throughput gains and should not really be considered. Generics are an excellent productivity tool and Ada designers should not be restricted from using them. However, it would be extremely wise to look at the code that is generated by your specific compiler. This author has used one Ada compiler (no longer validated) whose generic implementation was extremely inefficient. When using this compiler, generics were not even used.

Removing procedure calls in Ada is much easier than in other languages. The pragma inline feature allows the designer to improve timing (at the expense of additional memory usage) without impacting the level of abstraction of a module or it's readability. Because most Ada compilers are fairly efficient in the way code is generated for procedure calls, this alternative offers the greatest return for highly repetitive modules.

Floating point calculations can place a significant load on any processor (or processor/coprocessor pair). Very careful consideration should be given to selecting data types that are absolutely appropriate to their intended use. If it possible to sensibly convert a floating point data type to an integer type then, by all means do so. However, beware of using long integer types without carefully considering whether or not they are actually needed. There is at least one Ada compiler around that produces long integer code that runs 4 times slower than floating point code! Converting floating point types to fixed point is also an alternative that should be evaluated. This alternative requires careful consideration because of scaling, range and code readability implications. Moreover, there is one Ada compiler that implements fixed point types by generating floating point code.

A very positive approach to improving throughput is to restructure or redesign the algorithm that is driving the implementation. Once again, the concentration should be on those modules that are highly repetitive in nature. This is, perhaps, this author's preferred approach. This approach can force new insight into a requirement, engender creativity, and generally provide the best opportunity to flex one's software engineering/computer science muscles.

The alternative of transparently modifying the hardware is always extremely attractive, to the software team at least. This approach provides across-the-board throughput improvements and is usually feasible at later stages in a project as new and faster pin-compatible components become available.

Finally, the least desirable but generally most productive alternative throughput-wise, is the convert from Ada to assembly language. For those modules that are converted to assembly language, it is recommended that the original Ada module be continually maintained. In this way, as was mentioned above, there is a "functional" pseudo-code module that acts as an ultimate piece of documentation for the module.

As each timing improvement and its associated measurement is made, the throughput tables should be updated. Then the overall throughput of the software can be remeasured or calculated. When the desired overall throughput is achieved, the conversions can end. The objective is to meet the throughput requirement while at the same time retaining the largest amount of Ada.

In summary, the following steps should be followed to implement a hard real-time system using the Ada language.

1. Develop an initial timing estimate and a throughput budget.
2. Initially develop all software in Ada.
3. Perform timing measurements of all modules.
4. Assess which modules are most likely to cause throughput problems.
5. Evaluate alternatives for the problem modules.
6. Implement the appropriate method for improving "slow" modules.
7. If the improvement method involves conversion to assembly language, then maintain the original Ada code of the converted module.

8. Deliver both Ada and assembly language for converted modules.
9. Deliver Ada for unconverted modules.

William L. Miller
Senior Research Scientist & Branch Head, Engineering Sciences Laboratory, Georgia Tech Research Institute,
Atlanta, GA 30332,
(404) 894-7068

Bill Miller has 17 years of commercial and military real-time software development experience. He has been with the Georgia Institute of Technology for 2 years and his current primary responsibility as a Branch Head is to lead a software project that is integrating multiple EW subsystems using a 1750 processor, with software written in Ada, as the Mission Control computer. Previously Bill was with Rockwell International as the Manager of Software Design at Missile Systems Division, and lead the development Ada software for the HELLFIRE missile. His prior assignment was the Manager of Real-Time Software Engineering for the Strategic Defense and Electro-Optic Division, guiding the development of multi-processor, real-time image processing and signal processing applications. Bill has a B.S. in Computer Science from the University of California at Irvine and an M.B.A. from West Coast University.

Evaluation of Real-Time Distributed Systems Using Ada Concurrency

M. Bassiouni and *M. Chiu*
Computer Science Department

J. Thompson
Inst. for Simulation and Training

University of Central Florida
Orlando, FL 32765

Abstract

Rec:nt breakthroughs in computer and communications core technologies have made possible the interconnection of large number of real-time training devices via local area networks. The effectiveness and degree of realism achieved by team-training via distributed simulation are greatly influenced by the choice of the network topology and protocol used to interconnect the simulation devices. In this paper, we describe the design approach used to build Ada models for the performance evaluation of network protocols suitable for real-time distributed simulation. The paper describes the structure of the Ada software, highlights the important task inter-communication and synchronization aspects used in the design of the models, and presents examples of the performance results obtained via these models.

Key Words: distributed simulation, Ada concurrency, network protocols, real-time training systems.

1. Introduction

Recent breakthroughs in several computer and communications core technologies have made possible the interconnection of large number of real-time simulators (special purpose hardware) via local area networks. Two main applications/advantages of such networks are:

(1) To provide a low-cost effective tool for the training of personnel in applications involving interactions among mobile vehicles. Examples of such applications include training exercises for police forces, fire/ambulance services, and military combat fighting.

(2) To provide an effective "test before you build" development tool to be used for evaluating proposed modifications in existing systems, as well as an aid in designing/developing new systems. Tactics and coordination strategies might also be simulated and evaluated before they are adopted in real-life.

We shall use the terms "distributed simulation", "simulation networks", and "real-time training networks" interchangeably to denote the networking of a large number of real-time simulators for the purpose of training [6]. Each simulator consists of specialized hardware (a high-speed microcomputer, computer image generation subsystem, and sensor/control devices) bearing resemblance to the interior of the simulated vehicle (e.g., tank or police car). Each simulator has its own local copy of the database describing the simulated environment (e.g., city streets, buildings, terrain). As the crew of the simulated vehicle operate as they would in the real-life vehicle, the appropriate visual scenery is displayed on the CRT screens of their vehicle, as well as those of other vehicles in its sight range. It is obvious that the simulators participating in a training session must communicate with each other while carrying out the simulation. It is the responsibility of the underlying local area network (LAN) to provide each simulator with a reliable and fast mechanism to send and receive the information pertaining to the simulated activities.

The networking of real-time interactive simulation training systems departs from the traditional use of a computer network, whose function would normally be to provide sharing of computing resources among multiple users (nodes) on the network. When used to interconnect real-time simulators, the network is used almost exclusively for communication of process state information between the simulators engaged in the training exercise.

The Institute for Simulation and Training (IST) at the University of Central Florida has established a Network and Communications Technology Laboratory (NCTL/IST) dedicated to performing research for the purpose of enhancing the networking capabilities of distributed simulations. This laboratory houses a number of real-time simulators (different types of simulated ground and air vehicles) and is the center of several research projects dealing with the various aspects of real-time distributed simulation. In this paper, we describe the design approach used to build Ada models for the performance evaluation of network protocols suitable for real-time distributed simulation. The Ada models have been developed to provide a tool that can predict the performance of simulation networks and to gain valuable insight into the problems associated with the interconnection of real-time simulation devices. The models are also being used to complement the experimental network testbed at NCTL/IST.

2. Network Configuration Models

Various choices exist for the implementation of a LAN [1, 2, 3, 5, 7] (e.g., transmission medium, topology, access protocols, etc.) to interconnect simulation devices. In this paper, we limit our discussuion to token-based network protocols. Specifically, we describe a simulation approach using Ada which is suitable for token-ring and FDDI local area networks. The token ring configuration has a loop topology and employs a non-contention protocol that avoids collision by a token-passing mechanism [3, 4]. Figure 1 gives a block diagram of the basic configuration of the token-ring LAN. Simply stated, a token-passing ring is a LAN with a loop topology in which a token (a unique bit sequence in a data packet) is passed around the network, in a round-robin fashion, from one node to the next. Contention for transmission is resolved by stipulating that only the node currently in possession of the token is allowed to transmit a packet, or a sequence of packets, onto the ring. When the transmission is finished, the token is passed to the node downstream which then gains the right to transmit. Since there is a single token on the ring, only one node can be transmitting at a time. Other (non-transmitting) nodes, however, continuously receive the bit stream, examine it and repeat it onto the network (i.e., place it on the medium to the next station). A station repeating the bit stream may copy it into local buffers or modify some control bits if appropriate. A prototype token-ring scheme for real-time simulators has been recently completed at NCTL/IST.

The architecture of FDDI specifies a dual (counter rotating) token ring configuration that provides point-to-point connections between every pair of adjacent nodes on the ring. In normal operation, data packets circulate between the nodes using a single ring. The purpose of the dual ring is to gracefully recover from failures due to breaks in the transmission medium. Figure 2 shows a typical FDDI configuration that might be used to connect real-time simulation devices. Stations (i.e., simulators) are shown as nodes and connected in the form of a loop. Normally each simulator will be directly attached to the ring, but FDDI also allows the use of concentrators which can connect a number of stations to the ring. In simulation networks, these concentrators might be used to connect either a group of nodes which belong to the management and control of the training exercise, or to connect a group of regular nodes that are physically located near each other.

3. Properties of Networked Training

The application of networks to interconnect real-time simulators (for the purpose of training) has a

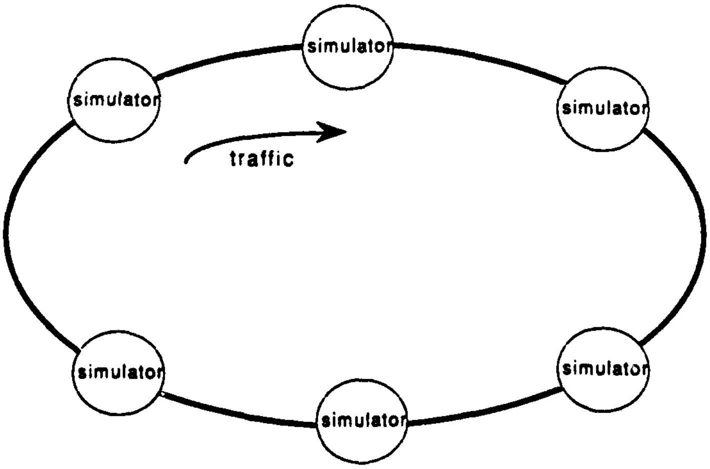


Fig. 1 Token ring network topology

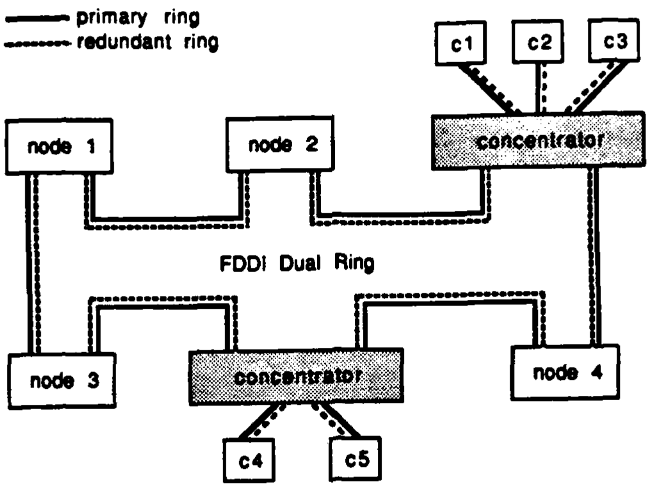


Fig. 2 An example of FDDI topology

number of characteristics and requirements. Recall that the main function of the LAN in this application is to communicate state update messages. When the state of a simulator changes (e.g., due to change in position or velocity, physical destruction, etc.) the simulator broadcasts a data packet of type "state update". This message is delivered by the network to every other simulator or node on the network. Upon receiving a state update message, each simulator updates its own local database and displays any appropriate changes on its screens. To accomplish this function under real-time constraints, the design of a simulation LAN must satisfy the following requirements:

(1) The network must provide connectionless data transfer services (datagram services) that include point-to-point transfer, multicasting, and broadcasting.

(2) The transmission delay incurred by a packet should be minimal [6].

(3) The percentage of lost packets should be kept as close to zero as practicably possible. The impact of lost packets on the fidelity of the training exercise depends on the type/contents of the packet. For example, the loss of a single state update packet from a slowly moving vehicle can be usually tolerated and would not much degrade the animated imagery displayed by other simulators. This is because the simulator of this vehicle will soon broadcast another state update message after a small time interval and, hence, its coordinates in the database of other simulators will be corrected.

4. The Ada Simulation Model

In this section, a high-level description of an Ada simulation model used in evaluating and predicting the performance of token-ring local area networks [3, 4] will be given. A similar simulation model has also been developed for FDDI LANs [1]. Since FDDI is primarily a token-based scheme, the basic design strategy described below for the token ring simulation model applies also to the FDDI counterpart. The concurrency mechanism in Ada has been used in our models to simulate the different concurrent activities within a simulation (training) network. A task in Ada may have entries which can be called by other tasks. Synchronization between two tasks occurs when the task issuing an entry call and the one accepting it establish a rendezvous. Communication in both directions is achieved via the parameters (input parameters) passed to the task accepting the entry call and those (output parameters) returned to the task issuing the entry call. This powerful synchronization facility has provided us with a convenient and elegant tool for modeling the parallel activities of the simulation network and the underlying networking protocol. The process interaction model of Ada has been used in our simulation to map the different entities and activities of the simulated network to corresponding Ada tasks. The following task types are the major generic entities used in the simulation of token ring LANs. Fig.3 depicts the interactions among these different tasks

1. *Source task:* is used to represent a vehicle simulator on the network. A task of this type is created for each such simulator

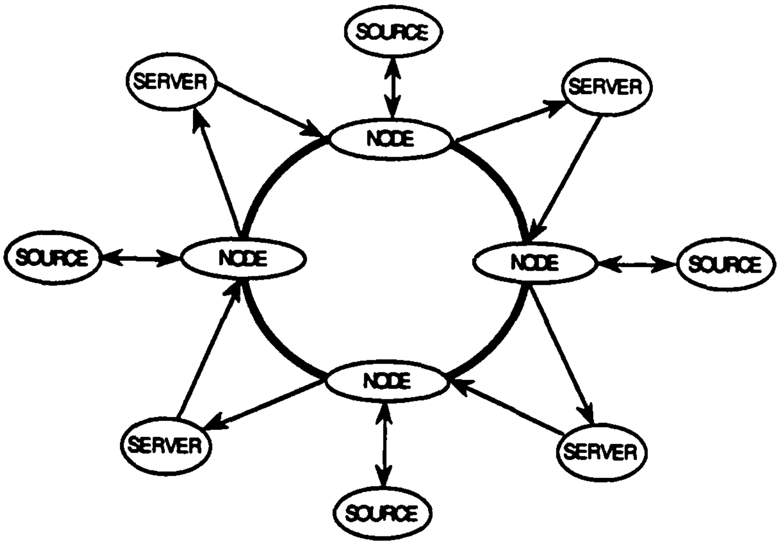


Fig. 3 Simulation model for token ring LAN

2. *Node task*: is used to represent the point of contact of each network node with the ring medium. It performs the functions of the medium access control(MAC) layer protocol and ensures that the token-ring protocol is executed. A task of this type is created for each network node.
3. *Server task*: is used to implement and control the flow of data on the ring. A task of this type corresponds to one of the point-to-point connections of the token ring LAN.
4. *Scheduler task*: (not shown in Fig. 3) is used to order timed events and control the sequencing of activities of the entire simulation.

In Ada, a task is composed of a specification and a body. The specification declares entries in which the data type and the input/output status must be specified clearly for each parameter. The body is the code that defines the activities of the task. Below, a brief functional description of each task is given. The description is written in pseudo code and only aspects related to the synchronization of the parallel activities within a token ring simulation LAN are considered.

Source Task

In this task, local traffic is generated according to a specified input method (e.g. using traces of real data or random stochastically generated inter-arrival times such as exponential, uniform, fixed with jitter, etc.). Whenever there is a packet in the queue, the Source task makes a request via an entry call to its Node task to transmit the packet to the network. The request may be blocked until the packet is accepted by the Node task.

```
-- task specification --
task type source_type is
    entry input (id, pktlength: in integer;
              sim_t, mean_iit: in float);
end source_type;

-- task body --
task body source_type is
     accept input (id, pktlength: in integer;
              sim_t, mean_iit: in float) do
    -- get input parameters
    end input;
    -- subscribe to the scheduler task
    sched.addUser;
    -- become a producer of the Node task
     nodes(my_node).addProd;

  -- main processing phase --
    -- get arrival time for the first packet
   t := erand (meanIit);
    -- request delay to wait for arrival
    sched.reqDelay (t, my_id);
    -- transmit the first packet
    nodes(my_node).transReq(pktlength,t);
   while simulation time has not expired
  loop
     -- get the arrival time of next packet
     t1 := erand(meanIit);
     sched.now(time)  -- get current time
     if time < t+t1 then
         -- wait for arrival of packet
           sched.reqDelay(time-t-t1,my_id);
     end if;
     -- update the packet arrival time
     t := t + t1;
     -- transmit this packet
     nodes(my_node).transReq(pktlength,t);
   end loop;

    -- termination phase --
   -- no longer a producer of Node task
    nodes(my-node).dropProd;
   -- no longer a user of  Scheduler
   sched.dropUser;
end  source_type;
```

Node Task

The Node task acts like a server ready to accept entry calls from its Source task or its neighboring Server tasks. When a packet from the producer (upstream) Server arrives, the Node task examines the type of the incoming packet. If it is a token, the Node task checks whether there is a pending local packet and if so, appends the free token at the rear of the packet and make them available to the consumer (downstream) Server. If it is a message packet, the Node may absorb it or make it available to the consumer Server depending on whether this packet was locally generated.

```
-- task specification --
task type node_type is
    entry input (sim_t, trans_rate: in float;
            FT_length, CT_length: in integer);
    entry addProd;
    entry dropProd;
    entry addCons;
    entry dropCons;
    entry transReq (length: in integer;
                start_t: in float);
    entry putmsg (nodeItem: in nItem);
    entry getmsg (item_t: out nItem);
end  node_type;

-- task body --
task body node_type is
    -- set up parameters
    accept input (sim_t, trans_rate: in float;
             FT_length, CT_length: in integer);
    end accept;
    sched.addUser; -- subscribe to scheduler
    sched.passive; --  initial state
    -- wait for first producer
    accept addProd do
      increment  producer  count
    end addProd;

   -- main processing phase --
    while there are clients loop
     select
      when Ok_local_packet =>
          accept transReq (length: in integer;
                     start_t: in  float) do
            -- get a new local packet
            -- make it a pending packet
          end transReq;
      or
        accept putmsg (Item: in nItem) do
           -- get  token/message-packet
          if msg is the free token FT then
            begin
              record the token cycle time
              if there is pending packet then
              append FT  to its rear; endif;
              msgReady := True;
            end
          else  -- message is a packet
             begin
              if packet is mine then absorb it
              else  msgReady := True;
             end
        end putmsg;
      or
       when msgReady =>
       -- unblock consumer server
         accept  getmsg (Item: out nItem) do
         if this was my pending packet then
              Ok_local_packet := True;
         end getmsg;
      or
       accept addProd do
          -- increment producer count
       end  addProd;
      or
       accept addCons do
          -- increment consumer count
       end addCons;
      or
       accept dropProd do
          -- decrement producer count
       end dropProd;
      or
       accept dropCons do
          -- decrement consumer count
       end dropCons;
      end select;
    end loop;
     -- print statistics --

   -- termination phase --
     sched.dropUser -- withdraw
end  node_type;
```

Server Task

The Server Task is used to implement and control the flow of data in the network. Each Server task takes the packet from its producer (upstream) Node and delivers the packet to its consumer (downstream) Node. The Server task also keeps track of the progress of propagation of the packets that it has transmitted and which have not yet arrived at the downstream node.

```
-- task specification --
task type server_type is
    entry input(id: in integer; sim_t,
      prop_time, trans_rate: in float);
end server_type;

-- task body --
task body server is
  -- initialization phase --
    accept input (id: in integer; sim_t,
      prop_time, trans_rate: in float) do
    end input;
    -- request to be a user of Scheduler
    sched.addUser;
    -- inform adjacent nodes
     nodes(upstream).addCons;
    nodes(downstream).addProd;

  -- main processing phase --
    while simulation time has not expired do
      -- request packet from producer Node
      nodes(upstream).getmsg(item);
      -- let ttrans be the time remaining for
      -- the transmission of current packet
      -- let tpropag be the remaining time
      -- until first traveling packet arrives
      -- at downstream node
     while  tpropag < ttrans  do
          begin
            sched.reqDelay(tpropag,my_id)
           nodes(downstream).putmsg(item);
            update ttrans and tpropag
          end;
      sched.reqDelay(ttrans,my_id);
   end loop;

  -- termination phase --
     nodes(downstream).dropProd;
    nodes(upstream).dropCons;
    sched.dropUser;
end  server_type;
```

Scheduler Task

The Scheduler maintains a list of all the tasks that make delay requests and reactivates these tasks at the proper time. When there is no active task in the system, the Scheduler reactivates the task at the head of the list and advances the clock accordingly. A task may be in the waiting state, in the active state, or in the passive state. The latter state is used when a task is blocked indefinitely, for instance, when a Server task requests to take a packet from its producer node but the packet is not available.

```
-- task specification --
task sched is
    entry now (ctime: out float);
    entry reqDelay (dt: in float; tid: in integer);
    entry addUser;
    entry dropUser;
    entry passive;
    entry active;
end sched;  -- specification

-- task body --
task body sched is
  -- initialization phase --
   -- accept the first user of Scheduler task
   accept addUser do
      -- increment client and active count
   end accept;

  -- main processing phase --
   -- accept requests while there are clients
   while client count > 0 loop
    select
      accept addUser do
         increment client and active count
      end  addUser;
   or
      accept  dropUser  do
         -- decrement client and active count
      end  dropUser;
   or
      accept  passive  do
         -- decrement active count when a
         -- task enters passive state
      end  passive;
   or
      accept  active  do
         -- increment active count when a
         -- task   in the passive state
         -- becomes active
      end  active;
   or
      accept now (ctime: out float) do
         -- set ctime to current time
      end  now;
   or
      -- handle a delay request
      accept reqDelay (dt: in float;
                        tid: in integer) do
           -- decrement active count
           -- add  task to the waiting list
      end reqDelay;
    end select;

    -- when there is no active task and
    -- the waiting list is not empty
    if active count = 0 and list not empty
      advance clock
      increment active count
      reactivate and delete head of list
    end if;
  end loop;

  -- termination phase --
end sched;
```

In addition to the above entities, several auxiliary tasks/packages are utilized for the purpose of collecting statistics, functions definition, user interfacing and task dispatching, etc. The software system is implemented in a modular fashion with emphasis on *ease-of-modification and the use of* parameterized values that *facilitate the testing of* a wide range of network characteristics and the simulation of different load conditions and different network parameters.

5. Performance Results

The Ada simulation models have been used to gain insight into the performance of simulation networks under both the token-ring and FDDI protocols. The models have been used to predict the performance of these two schemes when used to support a large number of real-time simulators. In what follows, we give examples of the results obtained by these models.

For the token-ring model, the recreation of the "free token" onto the ring is assumed to follow the "early token release protocol". According to this protocol, the transmitting station (the one which removed the free token from the ring) recreates the free token and puts it onto the ring immediately after it finishes transmitting its packet. This protocol results in better LAN throughput and smaller packet delays than protocols that require the header (or the tail) of the transmitted packet to complete one cycle around the ring before the free token is recreated. Another factor affecting the performance of the token-ring scheme is the length of the free token. Although this length has been used as a variable in our various tests, the results reported in this section use a length of 24 bits (24 bits is the length used in many commercial token-ring implementations).

The FDDI scheme uses an underlying token passing mechanism with appropriate modifications to handle synchronous data. If the FDDI parameters (thresholds and initial timer values) are carefully chosen, the FDDI LAN operates smoothly without (or with very few) reinitializations. In this case, the performance of FDDI is basically that of a high speed token ring. Unlike ETHERNET [2] whose performance deteriorates at high traffic loads due to excessive packet collisions, the overhead of FDDI token management does not result in throughput decline when the traffic load on the ring increases. Figure 4 shows a typical

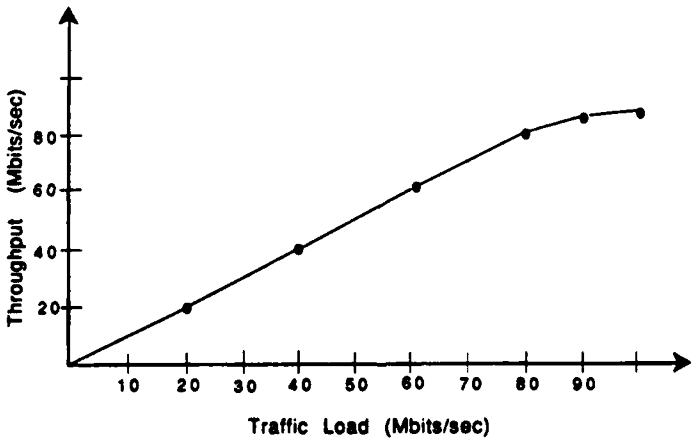
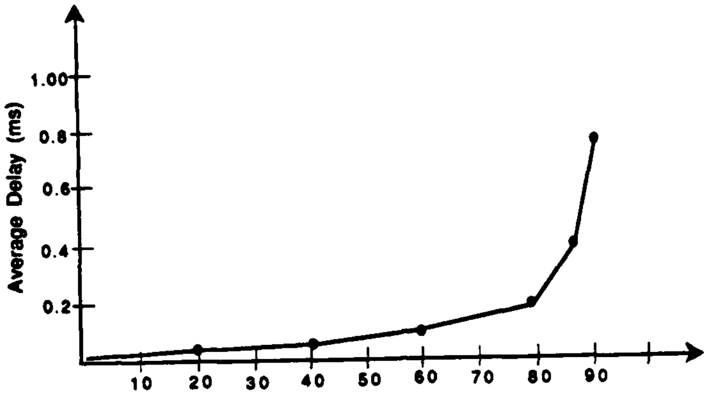


Figure 4



Traffic Load (Mbits/sec)
Figure 5

relationship between the throughput and traffic load in a token ring LAN. Since the token ring and FDDI protocols uses a collision-free scheme, they do not suffer from the problem of declining performance at high loads. Throughput around 90% of the transmission medium bandwidth can be easily obtained in token ring and FDDI LAN's (compare this to the ETHERNET protocol whose throughput is usually limited to about 65% of the medium bandwidth). Figure 5 shows a typical relationship between traffic load and the average time required for a packet to be successfully communicated through the network.

6. Conclusions

In this paper, we have described the high-level details of Ada simulation models used to evaluate the performance of networked simulation (training) systems under token based network protocols. The Ada models are also being used to perform a comparison study and evaluate different design decisions. Some of the numerical performance measures that are being gathered by the models are: the impact of traffic load on network throughput, the utilization of the transmission medium, and the distribution of delay times of transmitted packets. Further work is underway to use these models in evaluating schemes to incorporate real-time voice services within simulation networks.

Acknowledgements

This work has been supported by the Defense Advanced Projects Research Agency (DARPA) and the Army's Program Manager for Training Devices (PM TRADE) under Broad Agency Announcement #88-01, Contract N61339-89-C-0043. The views and conclusions herein are those of the authors and do not represent the official policies of the funding agency, the Institute for Simulation and Training, or the University of Central Florida.

References

[1] ANSI Standard X3.139 "FDDI token-ring media access control (MAC)" American National Standard, 1987.

[2] IEEE/ANSI Standard 8802/3 "Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specification" IEEE Computer Society Press, 1985

[3] IEEE/ANSI Standard 8802/5 "Token-ring access method" IEEE Computer Society Press, 1985.

[4] Dixon, R. ; Strole, N. and Markov, J. "A token ring network for local data communication" IBM system Journal, Vol. 22, 1983, pp. 74-62.

[5] Kummerle, K. and Reiser, M. "Local area networks- major technologies and trends" in .i "Local Area Networks," edited by K. Kummerle, J. Limb and F. Tobagi, IEEE Press, 1987, pp.2-27.

[6] Pope, Arthur "The SIMNET network and protocols" BBN Report No. 7102, BBN Communications Corporation, MA, July 1989.

[7] Stallings, W. "Local networks" ACM Computing Surveys, Vol. 16, No. 1, March 1984, pp. 3-42.

About the Authors

M. A. Bassiouni received his Ph.D. degree in Computer Science from Pennsylvania State University in 1982. He is currently an Associate Professor of Computer Science at the University of Central Florida, Orlando. His current research interests include computer networks, distributed systems, databases, and perforr ance evaluation. He has authored several papers and has been actively involved in research on local area networks, concurrency control, data encoding, I/O measurements and modeling, file allocation, and relational user interfaces. Dr. Bassiouni is a member of IEEE and the IEEE Computer and Communications Societies, the Association for Computing Machinery, and the American Society for information Science.

M. Chiu is a graduate student at the Department of Computer Science, the University of Central Florida. For the past three years, he has worked as a research assistant in the Communications and Networking Laboratory at IST/UCF. Mr. Chiu has played a key role in the design, coding and debugging of several simulation programs written in Concurrent C and Ada.

J. Thompson received his BS degree in Electrical Engineering from the University of Central Florida in 1978. He is currently a Research Associate for the Institute for Simulation and Training (IST) at the University of Central Florida, Orlando. Mr. Thompson has technical responsibility for all IST research activities involving computer and simulation networking.

A Dynamic Preference Control Solution to the Readers and Writers Problem*

Shakuntala Miriyala[†] *Tzilla Elrad*[‡]

The problem of mutual exclusion of several independent processes from simultaneous access to a *critical section* for the case where there are two distinct classes of processes known as *readers* and *writers* has drawn the attention of researches for several years. The *readers* may share the *critical section* with each other . but the *writers* must have exclusive access to the shared data. We present here. an Ada like solution to ensure no starvation of either of the processes. We do this by introducing what we call zero preference. A zero preference to an alternative is an indication to the compiler to ignore the alternative. Complicated solutions available for the above problem have been found difficult to implement. Therefore. the need for augmenting the available languages. with some constructs to implement different constraints arises. Here. we use dynamic preference control to solve the readers and writers problem. A solution to the same problem using static preference control has been presented in[5] .

1 Introduction

1.1 Motivation

The *readers and writers* problem has been a classical computer science topic for many years. A number of solutions to this problem in the Ada language have been suggested in the literature[1,4] . However, these solutions were found to be less acceptable. As they do not solve the problem of race between the two processes. The solutions presented here use preference control to solve the problem of giving preferences to the writing or the reading process. Preference control is not available in Ada. With the facility of preference control, the access to the shared data or the *critical section* can be controlled at compile time or run time depending on the input queue of *readers* and *writers*. There are different strategies adopted in solving this problem. The strategy adopted here is to be impartial to the *readers* and *writers*. Formally: *Many Readers or One Writer with the Writers Having a Higher Preference, But All Waiting Readers are Given Access after a Writer has Finished:* We however have split the reading and writing process into start_read, end_read, start_write and end_write.

Preference control is a race control at the task level which tells the compiler which alternative to accept when a conflict between alternatives arises. We claim in this paper that giving preferences to writers is a stronger requirement than avoiding writer starvation. The idea is that if a writer expresses a wish to write by entering the queue then the writing task should be given the highest preference. so that only the writing process is accepted. However, avoiding starvation of the writer simply means that the writer is accepted at some point of time. The main idea of this paper is therefore achieving a fair solution using dynamic preference control. Dynamic preference control is more efficient than the static preference control as the preferences change with the incoming scenario of the reader and the writer processes.

Preference control differs from the other form of race control namely the priority control in that priority is a race control at the program level which occurs outside the *select* statement in Ada. Regardless of the priority of the caller. preference controls the race between alternatives within a task.

1.2 Overview

In section 2 we discuss the need for preferences in solving the readers and writers problem. In section 3 the solution using dynamic preference control is presented. The classic Ada solution to the Readers

* This work is partially supported by the US Army Research CECOM and Battelle Scientific Services Program #1800

† Department of Computer Science. University of Illinois at Urbana-Champaign. Urbana. IL 61801.

‡ Department of Computer Science. Illinois Institute of Technology. Chicago. IL 60616.

and Writers problem available in literature is given in the appendix. Since all the solutions are in Ada we have avoided mentioning the finer details like package specifications , package definitions and the package body. We give only the task body of the controlling task the *reader-writer manager* and the package body of the whole package where necessary.

2 Solutions available in the literature

Before we present the existing solutions in the literature which deal with the reader and writers problem we would like to introduce some terminology.

Experience in controlling nondeterminism in Ada - an unconstrained choice from a finite number of alternatives has led to the following classification[2] :

- **Private Control:** nondeterminism restricted by variables local to the task. Private control is considered open if the boolean expression is true; it is closed otherwise. Only alternatives with the open private control are considered for selection by the nondeterministic construct.
- **Consensus control:** nondeterminism restricted by environmental/communication constraints. The choices are restricted to only those for which the communication request for the entry has been queued. Consensus control is considered established if the rendezvous can be established.
- **Hybrid Control:** nondeterminism restricted by both local boolean and environmental constraints. Hybrid control is available if private control is open and consensus control is established.

2.1 A solution using the count attribute in Ada

The solution using the count attribute to solve the *readers and writers* problem in[4] is *Solution 1* in the Appendix. Here we present some comments and drawbacks of the solution.

The use of a boolean guard WRITER_PRESENT is a reasonable evaluation criterion to determine the state of the local variables. Since the task executing the select statement is waiting for a call to one of the select alternatives, the value of these variables should not change. Thus it is reasonable to expect that the closed guard should remain closed and an open guard should remain open until the task performs some action. The use of a *count* attribute in a guard contradicts this expectation. Syntactically since the *count* attributes appears in the guard of a select statement the control is private. However, semantically the *count* attribute is consensus as its value depends on an activity that is not local to the present task. Moreover, the use of the *count* attribute does not solve the problem of race between alternatives. It has absolutely no control over the system behavior.

There are other implementation based issues involved when using a *count* attribute. One needs to make sure that the *count*, correctly reflects the number of items in the queue. A commonly occurring problem is when a reader enters the queue thereby causing the START_READ'COUNT value to be increased and later on decides to back out of the queue. while the value of START_READ 'COUNT remains unchanged. There is no guarantee that those readers who have entered the system are accepted. A count of available readers using the START_READ 'COUNT, accepts as many readers as indicated by START_READ 'COUNT. The compiler should take care of these problems using proper data structures.

There are solutions which avoid the use of the *count* attribute using static preference control in[illegible] . However, these solution do not concentrate on a fair solution (i.e,) avoiding the starvation of the reader and writer process. In the next subsection we provide a different solution using static preference control after explaining the need for preference control.

2.2 A Solution using static preference controls

Preference controls are race controls at the task level which tell us that when there is a conflict between two tasks which should be accepted. Each entry inside a nondeterministic construct may have a preference, which is an integer value of the predefined subtype *pref.* A lower value indicates a lower degree of urgency. The range of preferences is implementation defined. A detailed description of the syntax for introducing preference control in Ada is defined in[3] .

Ada language does not contain any primitive for controlling race between alternatives at the task level. Ada can use other available primitives to simulate preference control. The authors in[2] have argued that such a simulation of preference control using available primitives is very complicated and unacceptable by software engineering standards. Consider a resource manager, that continuously releases and regains resources. Since the manager must have a resource in order to release it, a wise manager should give preference to the regaining of the resources. The availability of a resource can be maintained by private control; the requests to release a resource can be checked by consensus control; but no primitive exists to express preference control, which is necessary in this case to give preference to the release of a resource.

We now classify preference control to be static and dynamic. Static preference control is to give the preferences in the program at compile time as shown below, whereas in dynamic preferences the preferences given to alternatives change with the dynamic nature of the problem itself.

Consider the following Controller which we call R_W_2 which has static preference control.

Solution 2:

```
task body R_W_2  is
   READERS: NATURAL := 0;
   WRITER_PRESENT:BOOLEAN := FALSE;
begin
   loop
    select

prefer 2:

      when not WRITER_PRESENT
       and START_WRITE'COUNT=0 =>
      accept  START_READ;
      READERS = READERS + 1;
      end START_READ;

 or
 prefer 1:
      accept END_READ;
      READERS = READERS - 1;
      end END_READ;

 or
 prefer 3:

      when not WRITER_PRESENT
           and READERS = 0 =>
      accept START_WRITE;
      WRITER_PRESENT := TRUE;
      end START_WRITE;

 or
 prefer 1:
      accept END_WRITE;
      WRITER_PRESENT := FALSE;
      end END_WRITE;
       for I in
         1...START_READ 'COUNT  loop
          --readers arriving after
          --the evaluation of the
          --count will not be
          --accepted here

         accept START_READ;
         READERS := READERS + 1;
         end START_READ;
        end loop
     end select;
   end loop;
end  R_W_2;
```

Package RESOURCE is the same as in solution 1 shown in the appendix. except that the task R_W_2 replaces task R_W_1 and calls to R_W_1 are replaced by calls to R_W_2.

In this solution we give highest preference to START_WRITE. After a writer is accepted all the waiting readers are accepted. This ensures that waiting readers are accepted immediately after a writer is accepted. Moreover we ensure that writers do not starve as we give preference to writers over readers. and do not accept the START_READ if START_WRITE 'COUNT is greater than zero. Here equal value of *prefer* means that any alternative can succeed.

Here we have introduced a mechanism for solving the race between two tasks by indicating at compile time which process should have a higher preference. However, We haven't really dispensed with the *count* attributes. We wish to point out here that the use of count attribute is a kind of availability control which if dispensed with will cause starvation. In the above solution if we remove the START_WRITE 'COUNT attribute the writers could starve if there was a scenario in which there is a continuous inflow of readers while the writer is waiting, as the value of READERS will never be zero. The above solution shows that by using a *prefer* construct in the code, an indication is given to the compiler to force to accept entry from writers first and then from the readers. Here the use of preference is to have a better control over the system behavior when more than one alternative is available. We thus argue that preference control is a much stronger requirement than the problem of starvation of writers.

The solution presented here differs from that in[5] in that here we have a *stronger requirement* that *none* of the processes should starve. In the solutions presented in[5] it is possible that reader starvation can occur. This is possible if there is a continuous inflow of writers and the local variable whcih keeps track of the readers is never incremented inspite of readers having entered the system. We argue that the need to get away with *count* attribute to ensure no conflict between syntax and semantics of the *count* attribute, is a trade-off for a fair solution.

Here, we have introduced a mechanism which will solve the race between tasks by explicitly giving preferences which adds to the efficiency of the solution to avoid starvation. The next logical question would be to have dynamic preference control making the preference control to vary in tune with the scenario of readers and writers. We present a dynamic preference solution to the *readers and writers* problem in the next section.

3 Solution using dynamic preference control

Here we present a solution that uses dynamic preference control which is more powerful than the static preference control. We introduce a concept of assigning a value zero to the *prefer* variable. If the value of the *prefer* variable is zero then it is an indication to the compiler to ignore the alternative following it. This causes less ambiguity when dealing with alternatives which are of no importance. Becasue we have indicated to the compiler exactly which alternatives have to be considered we have dispensed with the local variable "WRITER_PRESENT". For example to start with the value of prefer variable given to END_READ and END_WRITE alternative is not important.We therefore assign a value zero to them.

Solution 3:

```
task body R_W_3  is }
  READERS: NATURAL := 0;
  (SW,EW,SR,ER) := (2,0,1,0);
    -- Consider only start-read
    -- or start-write at the
    -- In case of race prefer
    -- start-write.
begin
   loop
     select

prefer SW:

    when READERS = 0 =>
    accept START_WRITE;
    (SW,EW,SR,ER) := (0,1,0,0);
       -- Accept only
       -- end-write; The rest
       -- are unacceptable.
 or

 prefer EW:
    accept END_WRITE;
    (SW,EW,SR,ER) := (1,0,2,0);
       --After end-write give
       --to readers. End-read
       --and end-write are not
       --available
 or
 prefer SR:
    accept START_READ;
```

```
     READERS := READERS + 1;
     for I
       in 1...START_READ 'COUNT
        loop
         accept START_READ;
         READERS := READERS + 1;
         end START_READ;
        end loop;
     if (START_WRITE 'COUNT > 0)
     then (SW,EW,SR,ER):=(0,0,0,1);
     else (SW,EW,SR,ER):=(0,0,2,1);
     endif
    end START_READ;
        -- If a writer is waiting
        -- initiate end-read. Else
        -- accept more readers.
   or
   prefer ER
      accept END_READ  do
      READERS = READERS - 1;
      end END_READ;
      if READERS = 0
      then (SW,EW,SR,ER):=(2,1,0,0);
      else (SW,EW,SR,ER):=(0,0,0,1);
      endif
        -- Complete reading and then
        --  accept start-write.

  end select;
 end loop;
end  R_W_3;
```

Initially either reading or writing can be done. If Start_WRITE is accepted then the only thing we can do next is END_WRITE. This is indicated to the compiler by giving a *prefer* zero value to all other alternatives. After END_WRTIE , which indicates the completion of writing process, we give highest preference to the reading process. Since we can allow multiple readers we allow as many readers waiting in the queue as indicated by the START_READ 'COUNT attribute. After reading it is logical to check for waiting writers and if there are, END_READ is given highest preference, if not, then more readers are accepted. And finally after END_READ, when all the reading is completed highest preference is give to the writing process.

Dynamic preference control allows us to alternate between the reader and the writer processes without causing starvation of either of the processes. We see that with dynamic preference control there is a better control of the calls accepted because, after each accept statement we know what best to accept next. This will be more efficient than static preference control. If we always gave a high preference to WRITE, and if there were no *writers* available, then we still have to check first for *writers* and then for *readers*. This is inefficient.

This solution still has an attribute which is semantically consensus. To do away with the attribute completely would be difficult as we are looking for a solution without starvation of either of the processes. We have shown through our solution that use of dynamic preference control is a good mechanism to solve the problem of races between tasks. A solution which uses pure private control and preferences may not completely solve our problem as there is no mechanism to control the behavior according to arrival scenario. This can be done only with the help of consensus control.

Preference control is presently not available in Ada. We can simulate static preference control in Ada, using nested select statements, but simulating dynamic preferences will be much more complicated and hence it may be simpler to augment the Ada language with preference controls.

4 Conclusion

Various solutions for the *readers and writers* problem where neither the *reader*, nor the *writer* starve were presented. We see that in all the solutions it was not possible to do away with the *count* attribute completely. The reason for this is that we are looking for a solution which would deal with the races between tasks without causing starvation of either process. To do away with the *count* attribute it would be necessary to provide availability control in an alternate manner. The purpose of this paper is to show how dynamic preference control improves the solution to the problem of races between tasks. We introduced the concept of zero preference. We achieved dynamic control by indicating to the compiler what alternatives are to be disabled or in other words indicating to the compiler to exclude those alternatives from race. This paper illustrates through the various solutions the inadequacy of Ada language for problems which involve race at the task level. The dynamic preference control principle can be extended to many other concurrent and distributed problems. It would be important to add preference controls to the existing concurrent languages as simulation of the same often leads to complicated and unreadable programs.

References

[1] Alan Burns. *Concurrent Programming in ADA*. Cambridge-University Press, Cambridge London, 1985.

[2] Tzilla Elrad and Fred Maymir-Ducharme. Introducing the preference control prmitive experience with controlling nondeterminism in ada. *Proceedings of the 1986 Washington Ada Symposium, Lurel, Maryland.*, 1986.

[3] Tzilla Elrad and Fred Maymir-Duncharme. Satisfying emergency communication requirements with dynamic preference control. *Proceedings of the sixth Annual Conference on Ada Technology, Arlington,Virginia*, 1988.

[4] Narain Gehani. *ADA Concurrent Programming*. Prentice-Hall, INC, Englewood Cliffs, NJ 07632. 1984.

[5] David Levin, Daniel Nohl, and Tzilla Elrad. A clean solution to the readers-writers problem without the count attribute. *National ADA technology conference*. 1989.

Appendix

Consider the following body for the controlling task named **R_W_1** which is a solution for the *readers and writers* problem using strategy 1[4] .

Solution 1:

```
task body R_W_1  is
  READERS: NATURAL := 0;
  WRITER_PRESENT:BOOLEAN:=FALSE;
begin
 loop
   select
     when not WRITER_PRESENT
       and START_WRITE'COUNT=0  =>
     accept  START_READ;
     READERS = READERS + 1;
     end START_READ;
   or
     accept END_READ;
     READERS = READERS - 1;
     end END_READ;
   or
     when not WRITER_PRESENT
       and READERS = 0 =>
     accept START_WRITE;
     WRITER_PRESENT := TRUE;
     end START_WRITE;
   or
     accept END_WRITE;
     WRITER_PRESENT := FALSE;
     end END_WRITE;
     for I in
      1...START_READ 'COUNT
        loop
           --readers arriving
           --after the evaluation
           --of the count attribute
           --will not be accepted
          accept START_READ;
          READERS := READERS + 1;
          end START_READ;
        end loop;
      end select;
    end loop;
  end R_W_1;
```

The package body is:

```
   package body RESOURCE  is

     S:SHARED_DATA := ...;
        --the shared data

     --specification and body
     --task  R_W_1 comes here

  procedure
   READ(X:out SHARED_DATA) is
  begin
   R_W_1.START_READ;
   X := S;
   R_W_1.END-READ;
  end READ;

  procedure
   WRITE(X:in SHARED-DATA) is
  begin
   R_W_1.START_WRITE;
   S := X;
   R_W_1.END_WRITE;
  end WRITE;

 end RESOURCE;
```

The RAPID Center Reusable Software Components (RSCs) Certification Process

Joanne C. Piper & Wanda L. Barner

U. S. Army Information Systems Software Development Center-Washington

ABSTRACT

Reusable code does not happen by accident. The effort to reuse a module includes obtaining the module, making changes to the module, and adding or adjusting a system of modules to make it work.[11] Through its certification process, the Reusable Ada Products for Information Systems Development (RAPID) Center minimizes, identifies, and isolates implementation dependent characteristics of Ada software, thus creating and promoting reusable code. The certification process includes defining the types of components for reuse, processing the components through RAPID's developmental phases, and providing the components a RAPID certification level.

INTRODUCTION

The RAPID Center's primary goal to promote Ada is accomplished by obtaining general purpose, adaptable software components that have maximum potential for reuse. The RAPID Center has established procedures and guidelines to certify and include Reusable Software Components (RSCs) in the RAPID Center Library (RCL). The RCL is an automated catalog and retrieval system that allows a user to identify and extract RSCs meeting specific functional requirements. An important part of the procedures has been the establishment of the RAPID Configuration Control Board (RCCB) which monitors the process and approves an RSC for inclusion in the RCL. The RCCB consists of the senior representatives from Quality Assurance, Configuration Management, Ada Engineering, and RCL Operations. The Final RAPID Center Reusable Software Component (RSC) Procedures, written by RAPID's senior Ada engineer, has been the step by step guide for the certification process. In order for the process to occur, RAPID has identified three stages: defining the RSC, developing the RSC, and certifying the RSC.

REUSABLE SOFTWARE COMPONENT DEFINITION

To fulfill the needs of a RAPID Center user, Domain Analysis is performed for the user's Standard Army Management Information System (STAMIS). Reuse opportunities are determined and the reusable components matching RAPID's definitions are identified and obtained. An RSC was originally defined as a source code component consisting of functions, procedures, or packages. Because code is not the only reusable software product, the definition was expanded to include five other components: requirements, design, implementation, templates, and generic architectures.

Requirements which are abstract ideas, identify the operating environment, the user's needs and tasks, and the data (objects) to be incorporated by the system. They are stated in a vocabulary that is common to all readers. Often the they are presented in the form of a Preliminary Design Review.[1]

Design components include compilable specification, logical data models, functional description documents, structure diagrams, data flow diagrams, and existing user's manuals.

Implementation components include package specifications, package bodies, generics, subroutines, subsytems, systems, and interactive test suites.

RAPID has not formally defined reusable Templates and Generic Architecture components. It has been determined that it would be best to presently concentrate on populating the RCL with low level software components.

DOMAIN ANALYSIS GROUPS

Domain Analysis further groups each previously defined type of component according to the level of abstractions or functional categories. The level of abstractions provide the perspective for the search of reusable components. The functional categories provide the actual list of RSCs to obtain. RAPID uses three levels of abstractions--Basic Software Engineering Level, Tool Level, and Application Level.[9]

The Basic Software Engineering Level provides subroutines of the common software building blocks (stacks, queues, linked-lists, trees) and functions (input validation, mathematics, string manipulation, file management, etc.).[9]

The Tool Level may include source code generators, form generators, compilers, linkers, and loaders that are functionally useful for processing.

The Application Level includes standard operations, such as sequential file transaction updates.

REUSABLE SOFTWARE COMPONENT DEVELOPMENT

During the development stages, the required RSCs determined by Domain Analysis are placed on the High Demand Category list. It is used to identify and evaluate components.

CATEGORY LISTS

Domain Analysis categorizes the RSCs that are grouped functionally by a system's architecture. The list driving the population of the RCL thus far has consisted of those RSCs that are basic software building blocks. A sample category list of other desirable MIS components include but are not limited to:[12]

Screen/Window Management
Interfaces for Common
Platforms
Forms
Communications/Networking
Tables
Reports
Access Security/Validation
Suspense/Schedule Management
Data Dictionaries
Relational Database Management
Ada-SQL Binding
Date/Time Conversion
File Maintenance
External Entity Interface
Management

IDENTIFICATION

A category list is established for each user of the RCL. RSCs are then obtained by the RAPID Center from public domain, Commercial-off-the-Shelf (COTS), and government sources. There are three names that identify the current state of an RSC--Proposed RSC, Candidate RSC, and RSC. Once the RCCB reviews and approves the Proposed RSC category list, a Software Development Folder (SDF) is created. The SDF contains printed forms for each certification process step--coarse evaluation, preparation, testing, documentation, classification, and maintenance. If it is determine during the coarse evaluation that to prepare the Proposed RSC is too costly, that RSC will be rejected by the RCCB but the SDF will remain on file.

COARSE EVALUATION

The coarse evaluation forms have been developed by RAPID's senior Ada engineer. The forms capture relative information determined during Domain Analysis and subjective visual analysis of the component.

1. A Reuse Potential Evaluation Worksheet contains the numeric ratings such as:[6]

a. Source Lines of Code - the physical lines of the source component including the blank lines, comments, and statements.

b. Effort to Reuse - the effort required to locate, extract, and use the RSC verbatim.

c. Effort to Produce - the effort required to develop a new component rather than reuse one.

d. Yearly Maintenance Effort - the average effort required to maintain the RSC for one year.

e. Uses captures the projected demand by the MIS domain categories identified on the Domain Analysis RSC Category form.

2. A Virtue Evaluation Form records the RAPID's Ada engineer subjective views of the RSC's characteristics. They are determined by reviewing the design, code or any related documentation:[6]

a. Known demand - the Proposed RSC has been specifically requested by a customer.

b. Short lead time - the RSC can be prepared and installed on the RCL with very little effort.

c. Wide applicability - the Proposed RSC can be used in a wide variety of applications.

d. High visibility - the Proposed RSC is targeted for a well-known application.

e. Compatibility - the Proposed RSC satisfies customer's needs.

f. Low complexity - the Proposed RSC can be easily understood and reused by the customer.

g. Drudgery avoidance - the Proposed RSC is necessary and well-known, but would be tedious to implement.

h. General appeal - the Proposed RSC demonstrates new technology, exceptional performance, well-designed user interfaces, etc.

3. High-Demand Evaluation Form - references categories established by Domain Analysis or the level of abstraction.

4. A preliminary metric evaluation is done for counting the number of source lines and determining quality metrics.

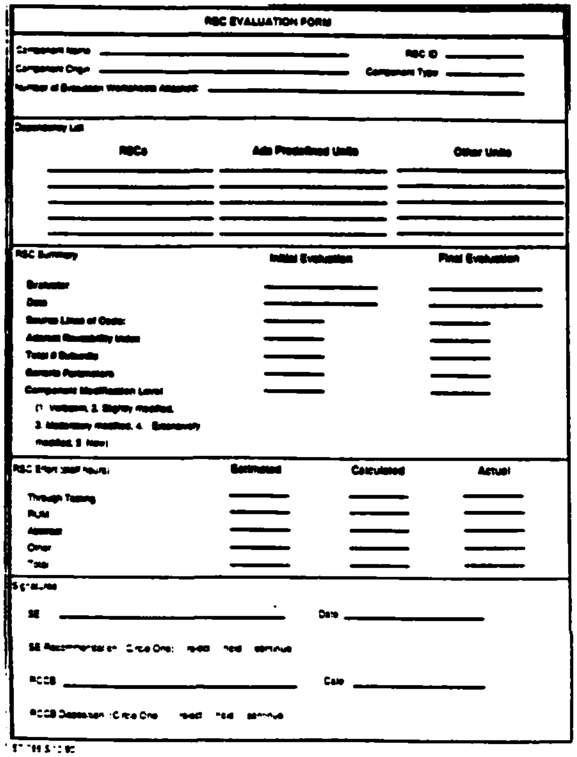
a. The RAPID Center's Source Lines of Code (SLOC) tool is used to automatically total the comments and non-commented lines for the software component.

b. Ada Measurement and Analysis Tool's (AdaMAT) metric aggregates, metric elements, and data items relate the software engineering principles by hierarchy to specific capabilities of the RSC.[4] The RAPID Center's reusability criteria is defined in the Information Systems Engineering Command(ISEC) Reusability Guidelines and The Information Systems Engineering Command (ISEC) Portability Guidelines. These published standards have been mapped to the AdaMAT's software quality criteria and metrics that pertain to non-complex understandable attributes of source code, attributes of high cohesiveness, understandable attributes of program structure, and attributes that detect machine dependencies.[5] Figure 1 represents the thresholds of percentages and counts allowed for the 31 metrics used to determine RAPID's Reusability Measure.

ANOMALY_MANAGEMENT	Normal_Loops	95%
	Constrained_Subtype	80%
	Constrained_Numerics	90%
	Constraint_Error	0
	Program_Error	0
	Storage_Error	0
	Numeric_Error	0
	User_Exception_Raised	100%
INDEPENDENCE	No_Missed_Closed	0
	Fixed_Clause	100%
	No_Pragma_Pack	0
	No_Machine_Code_Stmt	100%
	No_Impl_Dep_Pragmas	0
	No_Impl_Dep_Attrs	0
MODULARITY	Variables_Hidden	100%
	Private_Types_And_Part	100%
	Private_Types_And_Constant	100%
	Limited_Size_Profile	0
	Simple_Blocks	100%
	No_Variable_Declarations	100%
SELF_DESCRIPTIVENESS	No_Predefined_Words	100%
SIMPLICITY	Express_To_Do_Boolean	95%
	Array_Type_Explicit	100%
	Subtype_Explicit	100%
	No_Labels	0
	GoTos	0
SYSTEM_CLARITY	No_Default_Mode_Parameters	100%
	Private_Access_Types	100%
	Single_Implicit_Type	100%
	Module_End_With_Name	100%
	Qualified_Subprogram	80%

Figure 1

5. The RSC Evaluation Form (Figure 2) - contains the RSC long name, origin, type and the RAPID Center Configuration Control ID. It also lists the dependencies for the RSC and captures the preliminary evaluation metrics. The estimated time to prepare the RSC for testing and documentation is recorded. This form records the RCCB approval for a Proposed RSC to become a Candidate RSC.



RSC EVALUATION FORM

Component Name ____________ RSC ID ______

Component Origin ____________ Component Type ______

Number of Evaluation Worksheets Attached ____________

Dependency List

RSCs	Ada Predefined Units	Other Units

RSC Summary

	Initial Evaluation	Final Evaluation
Evaluator		
Date		
Source Lines of Code		
AdaMAT Reusability Index		
Total # Defects		
Generic Parameters		
Component Modification Level (1 Verbatim, 2 Slightly modified, 3 Moderately modified, 4 Extensively modified, 5 New)		

RSC Effort (staff hours)

	Estimated	Calculated	Actual
Through Testing			
RUM			
Abstract			
Other			
Total			

Signatures

SE ____________ Date ____________

SE Recommendation (Circle One): reject hold continue

RCCB ____________ Date ____________

RCCB Decision (Circle One): reject hold continue

Figure 2

PREPARATION

If the RSC passes the coarse evaluation, it becomes a Candidate RSC to be processed for preparation. RSCs shall be consistently formatted in such a way that the logical structure of the program is apparent to the reader therefore facilitating reusability.[10] Re-engineering conforms the component to guidelines found in the RAPID Center Standards for Reusable Software. When re-engineering is completed, a final metric analysis is done by running AdaMAT a second time, using all 150 metrics to determine reliability, portability, and maintainability. Reliability is determined by metrics for Anomaly Management and Simplicity. Portability is reflected through metrics for Independence, Modularity, and Self Descriptiveness. Maintainability is determined by the metrics of Modularity, Simplicity, System Clarity, and Self Descriptiveness.

TESTING

Once the Candidate RSC has been conformed to RAPID Center standards, the RAPID Center tests the component for functionality. A test suite is created for each Candidate RSC implementation type. The test suites consists of an interactive test driver that prompts the user, saves test input, and generates expected test results. The developer is able to run the test suite in batch mode and differentiate the expected output file and actual output file.

DOCUMENTATION

Thorough documentation consisting of time logs, design diagrams, provided users manuals, and developed Reusers Manuals are kept on each Candidate RSC. The time logs record the amount of hours spent by the Ada engineer to prepare or re-engineer the component for reusability. This information is used by the RCCB to determine if it is feasible to make a component, not originally designed for reuse, reusable. AdaGEN, an automated engineering tool, is used to reverse engineer Ada code and create design diagrams. The following AdaGEN diagram describes the dependencies of an Ada Package CmndLine. The arrows indicate the call patterns to other available packages represented by the solid ovals. The cloud represents an unknown component.[13]

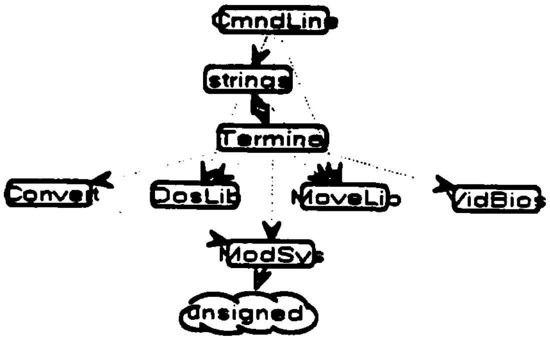


Figure 3

Users manuals serve as references for the development of the Abstract. Each abstract about two pages long is available on-line from the RCL and it describes the purpose of the component. Additionally, the Reuser's Manual which fully describes the development environment, generic parameters, dependencies, examples of operation for the Candidate RSC is written or vendor/developer supplied. Other documents that accompanied the component also are kept and made available through the RCL.

CLASSIFICATION

The RCL's automated search and retrieval system is based on a faceted classification scheme. Facets describe the views, aspects, or characteristics of the component. Classification is the last process on the Candidate RSC before it becomes an RSC installed in the RCL. RAPID currently uses nine facets to describe an RSC:

Description	Facet Name	Example Facet Term
conceptual abstraction	Object	Date
physical data structure	Data Representation	Character
performance process	Function	Convert
special methods	Algorithm	
software develop life cycle	Component Type	Implementation
methodological entity	Unit Type	Package
programming language	Language	Ada
approval level	Certification Level	Level 4
hardware/operating system	Environment	VMS/VĀX-Ada

Table 1

The example facet terms describes a function that converts dates represented by characters (i.e., July 1, 1990 to 07/01/90). The function is a package coded in Ada that runs under a VMS/VAX-Ada operating system. The Conversion uses no particular algorithm, so the algorithm field is not used.

An RSC may be described by multiple facet terms for the facets-- Object, Function, Data Representation, Algorithm, Unit Type, Language, and Environment. But, the facets Component Type and the Certification Level can have only one facet term.

MAINTENANCE

Maintenance, a continuous process of change, is the last phase of an RSC development state. Three sources contribute to the need for RSC maintenance: bugs, changes in the real world, and enhancement requests.[7] At the RAPID Center, configuration management and quality assurance are implemented for the life span of the RSC to ensure successful reuse and longevity. The Army Standard Engineer Change Proposal - Software (ECP-S) is the method use to record problems (bugs) encountered or enhancements suggested by RAPID's customer interaction and solicited feedback. An obsolete RSC is considered for the archive if it has not been reused and has no outstanding problem reports. RSCs are only deleted if an improved duplicate RSC is found, or an RSC's functionality is determined as being inherent to Ada.

REUSABLE SOFTWARE COMPONENT CERTIFICATION LEVEL

Originally RAPID had three levels of certification--tested, untested and certified. However, expanding population of the RCL dictated the need to refine RSCs certified as untested. There are now five RSC Certification levels and each level is progressive to the next. Levels 1 through 5 are defined as follows:[10]

Level_1 indicates the RSC has not been tested nor documented by the RAPID Center staff. But, an abstract, classification scheme, and metric analysis is prepared to install it on the

RCL. RSC's at this level are customer driven and would be placed in the RCL to meet immediate needs.

A Level_2 RSC has been reviewed by the RAPID Center staff and is expected to be reliable based on previous uses, even though testing of the RSC's functional performance has not been performed. The popular unguaranteed COTS software such as Booch-Wizard components and Government software such as AdaSAGE[6] apply in this category.

A Level_3 RSC has been reviewed and tested by either the RAPID Center staff or the supplying vendor/developer. The test materials are provided with the RSC. However, the vendor/developer components are not required to comply with RAPID's stringent standards for re-engineered components.

A Level_4 RSC has been reviewed and approved by the RCCB for compliance and/or deviations to RAPID standards of format, style, documentation, and complete test materials. COTS software such as EVB-GRACE[5] components that provide test scripts and reuser's manuals are classified at this level.

A Level_5 RSC is presently defined as having been prepared for Level 4 and it has been cleared for security purposes.

CONCLUSION

RAPID's program will increase user confidence by its services of catching the bugs, being aware of changes in the Department of Defense (DOD)/Industry Ada community, and providing configuration management of enhancements on all RSCs. Theoretically, these certified error free RSCs will last forever. In addition, the certification and preparation procedures and guidelines are continually being evaluated, refined and updated as the RCL continues to grow.

REFERENCES

1. Booch, Grady. Software Engineering With Ada. Canada: Benjamin/Cummings, 1983. 356 - 359.

2. Booch, Grady. Software Components with Ada. Canada: Benjamin/Cummings, 1987.

3. Dynamics Research Corporation (DRC). AdaMAT Product Information Sheet. Massachusetts: DRC, 1987.

4. Dynamics Research Corporation (DRC). AdaMAT Reference Manual. Massachusetts: DRC, 1987. 3-1, 4-1, 5-1, 6-1.

5. EVB. Generic Reusable Ada Components for Engineering Notes. Maryland: EVB Software Engineering, Inc, 1984-1989.

6. Idaho National Laboratory (INEL). AdaSAGE Reference Manual. Idaho: EG & G, Idaho, Inc, February 1990.

7. Infotech. Life-Cycle Management Analysis. State of the Art Report Series 8 Number . England: Infotech, 1980. 30.

8. SofTech. Final RAPID Center Reusable Software Component (RSC) Procedures. Massachusetts: SofTech, June 1990. 5-9, 6-7, C-6, H-22..

9. SofTech. User Orientation, Reusable Software Components Overview. Massachusetts: SofTech, June 1990. 301.22.51.

10. SofTech. RAPID Center Standards for Reusable Software. Massachusetts: Softech, October 1990. 3-6

11. Standard Automated Remote to Autodin Host (SARAH) Branch. Reuse of Ada Software Modules. Oklahoma: Command and Control Systems Office, June 1988.

12. Vitaletti, William and Ernesto Guerrieri, Ph.D. Domain Analysis Within the ISEC RAPID Center. Proceedings of the Eighth Annual National Conference on Ada Technology: 1990.

13. Mark V Systems Limited. AdaGen Users Manual. California: Mark V Systems Limited, 1990.

ABOUT THE AUTHORS

Joanne Piper is the Program Manger for the RAPID project at the Software Development Center Washington. She is responsible for the development and promotion of the RAPID program within the DOD/ Industry Ada community. Ms. Piper received her Bachelor of Science degree from the University of Maryland and is a member of SigAda and the National Association of Female Executives (NAFE). She currently chairs the Ada Joint Programming Office (AJPO) Reuse Initiative on management and DOD policy issues.

Wanda L. Barner is a Staff Sergeant in the United States Army assigned to the RAPID project at Software Development Center Washington. She is actively involved in the identification, procurement and certification of RSCs. She has specialized in application development through her eight years of computer experience. SSG Barner chairs the RAPID Configuration Control Board.

Both authors can be reached at:
USAISSDCW, ASQBI-IWS-R, STOP H-4, Fort Belvoir, Virginia 22060-5456.

SOFTWARE REUSE PROGRESS

James W. Hooper
Computer Science Department
The University of Alabama in Huntsville
Huntsville, Alabama 35899

Rowena O. Chester
Martin Marietta Energy Systems, Inc.
P.O. Box 2003
Oak Ridge, Tennessee 37831

ABSTRACT

In this paper we discuss and assess recent progr·ss in software reuse, as gauged by several indicators. The indicators fall into the categories of actual practice, research, and information dissemination.

In the category of practice, we consider reported results of reuse projects, including the extent of productivity improvements. We seek to identify common characteristics of successful reuse projects; both technical and managerial aspects are considered. We also assess the current extent of reuse practice, and seek to determine what applications are represented.

In the category of research, we review initiatives, approaches, and results. We seek to identify significant progress, and to indicate areas where further research is needed.

In the category of information dissemination, we summarize such indicators as workshops, conferences and short courses involving reuse, determine the nature of overview, tutorial, and guidelines material available, and summarize the status of journal articles on reuse.

In all categories, we attempt to identify developing trends. By providing this analysis of reuse progress, we hope to identify proven approaches for wider use. Much of the necessary material for this paper was drawn from the forthcoming book by these same authors, entitled SOFTWARE REUSE GUIDELINES AND METHODS, to be published by Plenum Press.

INTRODUCTION

Very significant progress has been made in the evolving field of software reuse. The first major hurdle overcome was the skepticism that reuse was feasible beyond "horizontal" reuse (meaning reuse of broadly-applicable software components such as mathematical and statistical routines, sorting routines, and the like). After the viability of "vertical" (i.e., application-specific) reuse was demonstrated in several projects, issues affecting large-scale reuse have been identified, and progress made in resolving them.

Availability of the Ada language has spurred interest in reuse, and Ada serves as the implementation language in many reuse projects. Growing emphasis on consistency of the software process has provided settings in which reuse can be effectively undertaken.

In the following sections we discuss, in turn, some achievements in practicing reuse, in conducting research into remaining reuse issues, and informing management and technical personnel about concepts and methods for software reuse. We have not attempted by any means to provide exhaustive coverage of current reuse status, but rather to discuss enough specific examples to convey a feel for status.

REUSE PRACTICE

Redundancy of software functionality has been recognized for some time. Jones (1984) estimated that of all code written in 1983, probably less than 15 percent was unique, novel, and specific to individual applications. However, only about 5 percent of code is actually reused, according to DeMarco (quoted in Frakes and Nejmeh 1987). Recognition of this disparity has spurred numerous companies and agencies to try to obtain benefit from previously developed software. It is recognized as being very important to try to obtain benefit from products resulting from all life-cycle phases, with the expectation that products from earlier phases should provide greater payoff if they can be effectively reused. While more is known at this time about reusing code than is true of other products, progress has been made in reusing products across the entire life cycle.

As we discuss different projects, it will become evident that the earlier reuse efforts were based primarily on ad hoc approaches, and that as greater understanding has been gained and research has begun to pay off, more systematic approaches are evolving. We now summarize a few experiences in reuse.

Selby (1989) studied software reuse activities at NASA Goddard Space Flight Center (GSFC). He considered 25 moderate and large-size software systems (from 3,000 to 112,000 lines of FORTRAN source code) that are support software for unmanned spacecraft control. The amount of software either reused or modified from previous systems averaged 32 percent per project. Selby characterizes reuse as a natural process in the GSFC environment--it is by developer choice rather than management directive, thus the developers must be convinced of the payoff. There is relatively low turnover of development personnel in this setting, and although there is variation in project functionality, the overall domain for all the projects is ground support software for unmanned spacecraft control, for which there is an established set of algorithms and processing methods. Thus reuse is facilitated by experienced personnel working in a stable, mature application domain. Subsequent efforts at GSFC with Ada code indicate reuse averages even higher than the 32 percent level experienced with FORTRAN code.

Lanergan and Grasso (1984) emphasize the importance of management commitment in Raytheon's successful reuse project. The Information Processing Systems Organization of Raytheon's Missile Systems Division concluded that about 60 percent of their business application designs and code were redundant. By standardizing those functions in the form of reusable functional modules and logic structures, they are experiencing about a 50 percent gain in productivity. Also, they report marked improvement in the maintenance process due to a consistent style for all software, which permits the reassignment of personnel from maintenance to development of new systems.

Biggerstaff and Perlis (1989b) reprint the Lanergan and Grasso paper as well as papers by Prywes and Lock (1989) and Cavaliere (1989) on reuse in business applications. Prywes and Lock used a program generator approach, with results of a threefold gain in programmer productivity. Biggerstaff and Perlis note that, while Cavaliere reports on an ad hoc approach at ITT Hartford Insurance, good results were obtained largely because "the Hartford management supported, capitalized, and actively moved to assure the success of the project." Biggerstaff and Perlis also reprint and comment on papers by Oskarsson (1989) and Matsumoto (1989), who report reuse successes in telephony software and process control software, respectively. They note that reuse skeptics doubted the possibilities of reuse in these domains, since the domains impose unusually strict memory requirements and performance constraints. Matsumoto, of Toshiba, states that on average about one-half of the lines of code of their generated software products are reused code. Biggerstaff and Perlis cite these experiences as reflecting "what can be accomplished with enlightened and committed management coupled with existing technology."

Prieto-Diaz (1990) discusses the approaches taken in the successful reuse project at GTE Data Services--called the Asset Management Program (AMP). The goal is to create, maintain, and make available at the corporate level, a collection of reusable assets. They define reusable asset as any facility that can be reused in the process of producing software; initial emphasis was on reusable software components. Based on the use of Prieto-Diaz's Faceted Classification approach for organizing a library of reusable components, the project was considered to be very successful during its first year. Although only 38% of the assets in the library were actively reused during the first year, a reuse factor of 14% was achieved--calculated by dividing the lines of code reused by the total lines of code produced by the organization. An estimated $1.5 million overall savings was realized. They were experiencing 20% reuse during the second year, and are predicting 50% reuse by the end of the fifth year, with a savings of over $10 million.

Reifer (1990) reports on the planning for a reuse program under the purview of the Department of Defense (DoD) Joint Integrated Avionics Plan for New Aircraft. The Joint Integrated Avionics Working Group (JIAWG) has made plans to address management, technical, and operational issues in the reuse program. Among issues addressed is a plan to incorporate specific reuse activities into each phase of DoD's software development process (as required by DOD-STD-2167A).

The DoD Strategic Defense Initiative Organization (SDIO), through its Software Reuse Committee, is also planning activities expected to lead to large-scale reuse within the Army and Air Force components, and their contractors.

Even the few examples given above indicate a wide range of application areas, organizations, and approaches. There is also a very noticeable acceleration of new starts in reuse, and they are becoming more ambitious as to organizational scope, life-cycle range of products, and up-front investment. We state some summary observations in the Conclusions section.

REUSE RESEARCH

A great deal of productive reuse-related research has been conducted, in the U.S. and abroad (especially in Europe and Japan). An influential early research project was the Common Ada Missile Packages (CAMP) project (McNicholl et al. 1986), conducted by McDonnell Douglas under contract to the U.S. Department of Defense (DoD) Software Technology for Adaptable, Reliable Systems (STARS) program. This was an early example of the viability of conducting vertical reuse, and also demonstrated the strength of Ada features for preparing reusable code components. STARS has also funded the development of a reusability guidebook (Wald 1986) and experimental reusable components, and now is undertaking a major reuse project with support from Boeing, IBM, and Unisys.

The federally funded Software Engineering Institute (SEI) at Carnegie-Mellon University in Pittsburgh has conducted several reuse research and experimentation projects, including use of the CAMP reusable parts, and is now conducting the Domain Analysis Project. Their goals are to develop domain analysis products that support implementation of new applications (including understanding a domain, supporting user-developer communication, and providing reuse requirements), and to establish domain analysis methods to produce these products. Hooper and Chester (1990b) discuss this project in detail.

U.S. Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), and its parent organization, U.S. Army Information Systems Engineering Command (ISEC), have both been active in reuse research. AIRMICS conducted the Ada Reuse and Metrics Project, with management by Martin Marietta Energy Systems, and funding from STARS. Hooper and Chester (1990a) summarize the results of the project, which included participation of researchers at a number of universities. ISEC is sponsoring the Reusable Ada Packages for Information System Development (RAPID) Center project, with support from SofTech; this project emphasizes the identification and retrieval of reusable Ada software components (Vogelsong 1989). A RAPID pilot project is presently underway, with the long-range goal being to expand usage to all of ISEC, and other agencies as need and funding allow. ISEC also funded preparation of reuse guidelines by SofTech (ISEC 1985).

The Software Productivity Consortium (SPC) in Reston, Virginia, conducts reuse research, including studies involving cost modeling for reuse assessment, relationships of prototyping to reuse, domain analysis, and other issues (Pyster and Barnes 1987). The Microelectronics and Computer Technology Corporation (MCC) in Austin, Texas, is conducting research in many facets of reuse, including the application of reverse engineering methods and hypermedia to reuse, and reuse of software components across the life cycle (Biggerstaff 1989). Numerous companies are active in reuse research and experimentation; in addition to those already mentioned, some others are Computer Sciences Corporation, Computer Technology Associates, CONTEL Technology Center, Draper Labs, GTE, Institute for Defense Analyses, Intermetrics, Rational, SAIC, and Westinghouse.

Significant research projects are underway in the United Kingdom, as evidenced by the special section on software reuse in the September 1988 issue of the Software Engineering Journal (Hall 1988); this issue contains some excellent papers. Gautier and Wallis (1990) document work conducted by the Reuse Working Group of Ada-Europe. The European Software Factory is a multination project to advance reuse knowledge/practice in Europe. The software factory approach is also receiving research emphasis in Japan (e.g., see Fujino 1987).

In addition to research projects, per se, much has also been learned about what is effective (and in some cases, what is not effective) in conducting actual reuse projects, as discussed in the section above. Assessment of research results is also occurring in actual reuse projects, and needs for additional research are being identified. So far research has been conducted, and results applied, in such diverse aspects of reuse as domain analysis, preparing reusable components, library mechanisms for categorizing components, storing and maintaining them, retrieving and understanding them, modifying them, and binding them together into workable software. Also, research has been (and is being) conducted in cost modeling for reuse, management and organizational approaches to foster reuse, and legal and contractual strategies for reuse. Human behavioral and cognitive aspects of reuse are also being studied. The software engineering process is being studied, since reuse must be an integral part of software development and maintenance if it is to be successful. Benefits of object-oriented methods are being assessed for reuse support. Various issues related to tool/environment support for reuse are being addressed, including knowledge-based approaches.

Thus, little-by-little, research is paying off in these different facets of software reuse. In the meantime, successful reuse is occurring based on methods already known to be effective. As new and better methods are devised, even greater productivity will no doubt result from reuse, and reuse will be undertaken in areas in which it is not now occurring. We make some further comments about the need for additional research, in the Conclusions section.

INFORMATION DISSEMINATION

As is true in the categories of research and practice, acceleration of activities is being experienced in the dissemination of information about reuse. And, as the earlier comments in this paper would suggest, the information being supplied is of a higher quality than was previously the case, and indicates increasing understanding of reuse and greater maturity of processes. Also, a growing confidence is being expressed in the viability and economic feasibility of reuse in carefully selected application areas.

Some excellent papers on reuse have appeared in refereed technical journals. The September 1984 issue of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING is a seminal reference source for early work. The July 1987 special issue of IEEE SOFTWARE, "Making Reuse a Reality", is another important source. Individual papers on both reuse experiences and reuse research are now appearing frequently in refereed journals.

Most of the conferences on software engineering and on Ada have a reuse track, with numerous papers being presented. Tutorials on reuse are being offered at many conferences, and short courses are appearing as well.

Three tutorials are available: Freeman 1987, Tracz 1988, and Biggerstaff and Perlis 1989a and 1989b. A forthcoming book by Hooper and Chester (1990b) overviews the field, and offers guidelines for both managerial and technical aspects of reuse--including suggestions for getting started in software reuse. Several earlier guidelines documents are available (e.g., ISEC 1985, Wald 1986). Also, several new software engineering textbooks include material on software reuse.

CONCLUSIONS

In all the categories of practice, research, and information, indications are that significant progress has been made in software reuse. Research has led to greater understanding and better practice, and success in practice has led to greater confidence in the viability of reuse, and to less reluctance to fund reuse projects. The indications are that in all three areas, the level of productive activity is steadily increasing.

A number of specific observations are offered here.

* The greatest growth in reuse is occurring in application- specific projects.
* Reuse of products other than code is progressing rapidly.
* Reuse projects are becoming more ambitious as to organizational scope and up-front funding.
* Reuse approaches are becoming more systematic/formal, less ad hoc.
* Successful reuse projects all have strong management committment.
* Successful reuse projects occur in stable, mature application domains with personnel experienced in the domain.
* Successful reuse projects are characterized by consistent development and maintenance processes.
* Successes are occurring in a wide range of application areas--business, scientific and engineering, both with and without difficult time and memory constraints.
* Effective research has been responsible for greater success in reuse.
* Research continues on various fronts--and is needed to further improve various aspects of reuse.
* Information sharing concerning reuse is increasingly effective.

An important area of current research is cost modeling to predict comparative costs to reuse, redevelop, and purchase, as well as predicting costs to make components reusable. A related need is for organizations to build up data on their own costs to develop reusable software, to develop software without reusability emphasis, to reuse software, and to modify software.

Legal and contractual issues need further study and resolution--relative to data rights and potential liability. The relationships of the U.S. government to its contractor presents especially difficult issues. Further study and experimentation is needed concerning the integration of reuse into the overall software process. Additionally, better domain analysis methods and tools are needed, as are more effective means to match software needs to available reusable software--involving a great range of technical disciplines.

The progress so far in software reuse is very encouraging, offering motivation for further expansion of both practice and research. Many interesting and challenging opportunities are before us.

REFERENCES

Biggerstaff, T. 1989. Design Recovery for Maintenance and Reuse. COMPUTER, vol. 22, no. 7 (July), p. 36-49.

Biggerstaff, T.J. and A.J. Perlis (eds.). 1989a. SOFTWARE REUSABILITY. VOL. I, CONCEPTS AND MODELS. ACM Press (Addison-Wesley, Reading, Mass.).

Biggerstaff, T.J. and A.J. Perlis (eds.). 1989b. SOFTWARE REUSABILITY. VOL. II, APPLICATIONS AND EXPERIENCE. ACM Press (Addison-Wesley, Reading, Mass.).

Cavaliere, M.J. 1989. Reusable Code at the Hartford Insurance Group. In Biggerstaff and Perlis 1989b.

Frakes, W.B. and B.A. Nejmeh. 1987. Software Reuse Through Information Retrieval. In PROCEEDINGS OF THE 20TH HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, Kailua-Kona,Hawaii, pp. 530-535.

Freeman, P. 1987. TUTORIAL: SOFTWARE REUSABILITY. The IEEE Computer Society.

Fujino, K. 1987. Software Factory Engineering: Today and Future. In PROCEEDINGS OF THE 1987 FALL JOINT COMPUTER CONFERENCE, Dallas, pp. 262-270.

Gautier, R.J. and P.J.L. Wallis. 1990. SOFTWARE REUSE WITH ADA. Peter Peregrinus Ltd. (London, U.K.).

Hall, P.A.V. (guest ed.) 1988. Software Components and Re-use: special section of SOFTWARE ENGINEERING JOURNAL, vol. 3, no. 5 (Sept.).

Hooper, J.W. and R.O. Chester. 1990a. SOFTWARE REUSE GUIDELINES. U.S. Army AIRMICS ASQB-GI-90-015.

Hooper, J.W. and R.O. Chester. 1990b. SOFTWARE REUSE GUIDELINES AND METHODS (DRAFT) (Sept.). (Accepted for publication by Plenum Press.)

ISEC 1985. ISEC REUSABILITY GUIDELINES, 3285-4-247/2. SofTech (for U.S. Army Information Systems Engineering Command; Dec.).

Jones, T.C. 1984. Reusability in Programming: A Survey of the State of the Art. IEEE TRANS. ON SOFTW. ENG., vol. SE10, no. 5 (Sept.), pp. 488-494.

Lanergan, R.G. and C.A. Grasso. 1984. Software Engineering with Reusable Design and Code. In IEEE TRANS. ON SOFTW. ENG., vol. SE10, no. 5 (Sept.), pp. 498-501.

Matsumoto, Y. 1989. Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels. In Biggerstaff and Perlis 1989b.

McNicholl, D.G., C. Palmer, et al. 1986. COMMON ADA MISSILE PACKAGES (CAMP). Vol. I: Overview and Commonality Study Results. AFATL-TR-85-93. McDonnell Douglas, St. Louis, MO.

Oskarsson, O. 1989. Reusability of Modules with Strictly Local Data and Devices--A Case Study. In Biggerstaff and Perlis 1989b.

Prieto-Diaz, R. 1990. Implementing Faceted Classification for Software Reuse. In PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON SOFTW. ENG., Nice, France (Mar.)

Prywes, N.S. and E.D. Lock. 1989. Use of the Model Equational Language and Program Generator By Management Professionals. In Biggerstaff and Perlis 1989b.

Pyster, A. and B. Barnes. 1987. THE SOFTWARE PRODUCTIVITY CONSORTIUM REUSE PROGRAM. SPC-TN-87-016, SPC, Reston, VA. (Dec.).

Reifer, D.J. 1990. JOINT INTEGRATED AVIONICS WORKING GROUP REUSABLE SOFTWARE PROGRAM OPERATIONAL CONCEPT DOCUMENT (OCD) (DRAFT). RCI-TR-075B, Reifer Consultants, Inc., Torrance, CA. (June)

Selby, R.W. 1989. Quantitative Studies of Software REuse. In Biggerstaff and Perlis 1989b, 213-233.

Tracz, W. 1988. TUTORIAL: SOFTWARE REUSE: EMERGING TECHNOLOGY. The IEEE Computer Society.

Vogelsong, T. 1989. Reusable Ada Packages for Information System Development (RAPID)--An Operational Center of Excellence for Software Reuse. In PROCEEDINGS OF THE REUSE IN PRACTICE WORKSHOP, Pittsburgh (July).

Wald, B. 1986. STARS REUSABILITY GUDEBOOK. V4.0. DoD STARS.

BIOGRAPHIES

JAMES W. HOOPER is Professor of Computer Science at the University of Alabama in Huntsville (UAH), where he teaches and conducts research in programming languages and software engineering. He holds B.S. and M.S. degrees in mathematics, and M.S. and Ph.D. degrees in computer science. Prior to joining UAH in 1980, he was employed by NASA Marshall Space Flight Center, where he conducted research in simulation approaches for NASA missions.

ROWENA O. CHESTER manages research projects in the Data Systems Research and Development Program of Martin Marietta Energy Systems, Oak Ridge, Tennessee. Recent research activities include operating system and application software certification, database security, PC vulnerability assessments, portability, and Ada software reuse. She holds the B.Eng. Physics degree, and the Ph.D. in physics and electrical engineering.

An Integrated Approach to Software Effort and Schedule Estimates

Basil Papanicolaou

Lockheed Sanders, Nashua, NH

Abstract

Software development effort estimates and planning for large and complex programs is a challenge. Several models are used to support this task, however they are not flexible to accommodate individual project needs and their results draw little confidence. This paper presents a comprehensive integrated approach that provides consistent and accurate estimates.by comparing first the characteristics of commonly used software parametric models, namely Ada COCOMO, REVIC, and PRICE S. The approach addresses current software development processes, such as the use of the Ada programming language, incremental development, and software engineering environments. A simple methodology has been established, where the same parameters feed different models, if required, and yield comparable results. This facilitates analyses and minimizes the need for result reconciliation, while it increases confidence.

1. Introduction

Presently many software developers encounter planning for Ada projects for the first time. While they are lacking their own cost experience for such projects they can rely on software parametric models that support Ada development. Since models differ, their features and outputs must be well understood and compared to engineering estimates as they are used.

This work represents recent experience in planning a large Ada project. It outlines an overall process where software parametric models are utilized.

Section 2 outlines software size estimation. Section 3 discusses factors to be considered for selection of software development approach. Section 4 presents effort and schedule estimation, including discussion of parametric models. And finally section 5 addresses software development plan provisions for staying within the initial estimates.

2. Software Size Estimate

The process starts by estimating the size of software in lines of code (LOC). It is usually done starting by top down functional or object oriented decomposition and estimate of the size of lower level components by comparison to known projects or by engineering estimates. A bottom-up summation to the subsystem and system level follows.

Software sizers are also available and they may be used if software size knowledge data base is not available.

3. Software Development Approach

The decision on the software development approach is based on project requirements: Product complexity, reliability, risk, cost, experience base, resource availability, customer schedule, etc. The objective is the development of quality software, that meets the customer requirements, within budget and schedule.

An initial set of all the tentative software development environment, covering programmatic, developmental resource, and system aspects, is defined. Subsequently software parametric tools will be used for trade-off analyses.

4. Effort and Schedule Estimates

The results of sizing and initial environmental parameter assessment are used as input to software effort and schedule estimates, which are performed through parametric models and engineering analysis.

4.1 Software Parametric Model selection

One or more parametric models may be selected for effort estimate, by customer or company requirement, or for more confidence. Most common models specifically developed or tailored for Ada development are:

- Ada COCOMO [1], based on COCOMO [2], which has been modified with the addition of Ada Scaling equation and calibrated with completed Ada programs. It is meant for programs following the Ada Process Model [3], however it can be applied to disciplined DOD-STD-2167A/ MIL-STD-1803 programs with very good approximation.
- REVIC [4], based on the original COCOMO and sponsored by the Air Force, and
- PRICE S [5], which is part of the GE PRICE family of tools.

4.2 Parametric Model Features

All three referenced models are supporting both software development and life-cycle. All three have been validated with respect to different sets of projects. Table 1 is a comparative chart of their features.

4.3 Inputs and Outputs

Table 2 shows the required input parameters for each model, besides size and schedule, as related between models. If more than one models are used, this table is used to assign compatible values to the models Figures 1 and 2 represent the results of uncalibrated models for the same activities of a typical 100,000 LOC Ada project. Notice the 2:1 effort discrepancy between PRICE S and Ada COCOMO. Effort and schedule distribution are shown in Table 3. Ada COCOMO and REVIC allocate more time for the early phases, typical characteristic of Ada software development.

4.4 Effort and Schedule estimate

Parallel with the models, which provide an initial prediction, the development effort is estimated using composit implementation predictors [6], that is intuitive engineering estimates for each software activity at the system and subsystem level, based on productivity achieved in similar programs within the same or similar environment. This process considers schedule constraints, resource availability, learning curves, and risk reduction, and leads to the generation of activity networks showing activity dependencies in a time scale.

The models, on the other hand, are calibrated with data from the developer's cost history data base, if available, or checked and normalized against engineering estimates. Ada COCOMO and REVIC are calibrated by adjusting the equation constants. PRICE S is calibrated by deriving a parameter representing developers productivity (PROFAC). Calibration reduces result discrepancy between models.

If risk considerations or customer requirements call for incremental development each increment is modelled separately. Parameters may vary between increments.

The models are then used at the subsystem level. Results are fed into the generated activity networks for critical path analysis. At this point their tradeoff analysis capability is very valuable.

Table 1.
Parametric Software Model Features

FEATURE	Ada COCOMO	REVIC	PRICE S
Ada Development Mode	Yes	Yes	Yes
Incremental Development Modelling	Yes	Indirect	Indirect
System/Software Requirements	No	Yes	Yes
System Integration Support	No	Yes	Yes
Detailed Financial Factors	No	No	Yes
Operator Friendliness	N/A	Very Good	Good
Access	Model Equations	Down-loadable PC program	Via modem
Support	User's Group	User's Group	800-Number
Cost	Free	Free	License, Connect, Storage fees

Table 2.
Software Model Environmental Parameters

REVIC	Ada COCOMO	PRICE S
Analyst Capability	Analyst Capability	Complexity1-Personnel
Programmer Capability	Programmer Capability	Complexity1-Personnel
Applications Experience	Applications Experience	Complexity1-Product Familiarity
Virtual Machine Experience	Virtual Machine Exp	Complexity1-Product Familiarity
Programming Lang Experience	Programming Lang Experience	Complexity1-New Language
Execution Time Constraint	Execution Time Constraint	Utilization
Main Storage Constraint	Main Storage Constraint	Utilization
Virtual Machine Volatility	Virtual Machine Volatility (Host, Target)	Complexity 2-HW dev at the same time
Computer Turnaround Time	Computer Turnaround Time	(PROFAC)
Requirements Volatility	Requirements Volatility	
Product Reliability	Required Software Reliability	Platform
Product Complexity	Product Complexity	Application
Data Base Size	Data Base Size	
Required Reuse	Required Reuse	
Modern Programming Practices	Modern Programming Practices	(PROFAC)
Use of Software Tools	Use of Software Tools	Complexity1-Software Tools
Required Security	Required Security	
Management Reserve for Risk		
Required Schedule		
SW Development Mode		
	Complexity with Respect to Personnel Capability	

Ada Scaling Equation
Experience with Ada Process Model
Design Thoroughness at PDR
Risks eliminated by PDR
Requirements Volatility

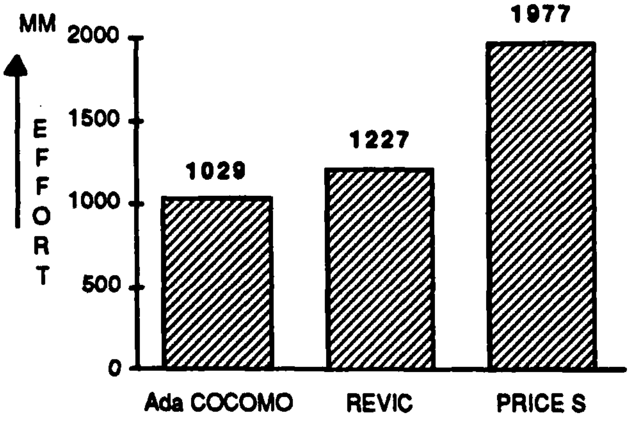


Figure 1. 100 KLOC Software Development Effort Estimate by Uncalibrated Models

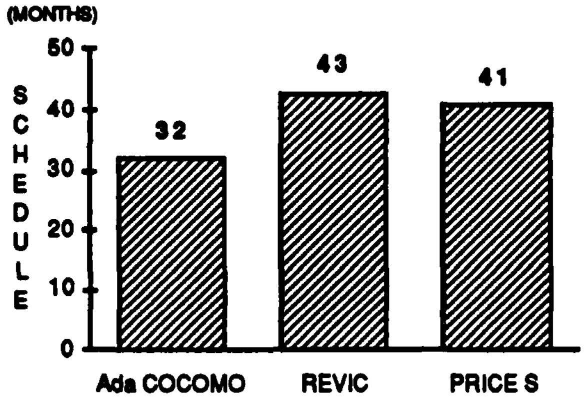


Figure 2. 100 KLOC Software Development Schedule Estimate by Uncalibrated Models

Table 3.
Parametric Model Effort and Schedule Distribution per Software Activity

	EFFORT (%)				SCHEDULE (%)			
ACTIVITY MODEL	PD	DD	CUT	SW I&T	PD	DD	CUT	SW I&T
Ada COCOMO, REVIC	23	29	22	26	39	25	15	21
PRICE S	18	28	25	30	20	23	33	24

Risk assessment and risk mitigation must also be part of the planning activity. AFSC/AFLCP 800-45 [7] is an excellent guide. Risk reduction activities become part of the software activity network.

Finally in the plan and cost considerations all project unique activities, such as special support, training, on-site support, etc, must be included. These tasks are not part of the effort covered by the models.

3. Software Development Plan

The effort and schedule estimates made in the planning phase assume that certain conditions will exist during development. The adherence to these conditions is measured through software management indicators.which should be defined and incorporated into the software development plans. An excellent set of management indicators is provided in AFSC/AFLCP 800-45 [8].

References

[1] Barry Boehm and Walker Royce, "Ada COCOMO: TRW IOC Version", Proceedings, Third COCOMO User's Group Meeting, Pittsburg, PA, November 1987.

[2] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[3] Walker Royce, "TRW's Ada Process Model for Incremental Development of Large Software Systems", Proceedings, 12th International Conference on Software Engineering, Nice, France, March 1990.

[4] REVIC User's Manual

[5] *PRICE S Reference Manual*, General Electric Company, 1989

[6] Tom DeMarco, *Controlling Software Projects*, Yourdon Press, 1982

[7] *Acquisition Management Software Risk Abatement*, AFSC/AFLCP 800-45, Air Force Systems Command and Air Force Logistics Command.

[8] *Software Management Indicators*, AFSCP 800-43, Air Force Systems Command.

BIOGRAPHY

Basil Papanicolaou is a software engineer at Lockheed Sanders. During his eight years at Sanders he has been part of software design and management teams. Prior to Sanders he worked at RCA, Burlington, and Dialog Systems, Belmont, Massachussetts.

Mr. Papanicolaou holds a BS in Physics from the University of Athens, and a MS in Physics from Thomas Jefferson University.

MANAGING THE Ada CONVERSION AND INTEGRATION OF MISSION CRITICAL DEFENSE SYSTEMS

Thomas S. Archer

TELOS Systems Group
Sierra Vista, Arizona

ABSTRACT

This paper discusses the importance of Project Management's role in effectively coordinating and integrating the goals and approaches of the user, customer, and support contractor during the development or enhancement of Mission Critical Defense Systems.

INTRODUCTION

The degree of success obtained in the development or enhancement of Mission Critical Defense Systems software has always been impacted by one constant management factor. That factor has been and is the measurable degree of Project Management's success obtained in coordinating and integrating the goals and approaches of the Government user, the Government customer/Project Manager, and the Development/ Support contractor. The Department of Defense selection and mandating of Ada as the language to be used for developing Mission Critical Defense Systems software and its subsequent use have increased the degree of success obtained in integrating the management goals and approaches. This increase has been proportional to the degree of incorporation of Ada's inherent rigidity and structured architecture into the project and its management approach. This paper in the next four sections will address: the selection and integration of management goals and approaches by the user, customer, and contractor; the Ada structural concepts being used and the support being provided management during development and enhancement of those systems; the management approaches being used to integrate Ada structures and constructs into the development of new or enhancement of existing systems; and the current approaches to incorporating of Ada into new or existing Mission Critical Defense Systems, their purposes, and resulting benefits.

MANAGEMENT SELECTION AND INTEGRATION

The development or enhancement of a Mission Critical Defense System is a project composed of a series of tasks that move the effort through system life cycle phases to delivery of the product. For the project to be successful, all participants must ensure that the direction, progress, time, and resources are managed efficiently and effectively throughout the effort. The first item to be selected is the Project Management organizational structure.

Organizational Structure Selection.

There are several elements that effect the selection of an organizational structure for development or enhancement of a Mission Critical Defense System. Some of these elements are the size of the effort, the type of development or enhancement, and current and projected organizational responsibilities. The Ada language supports all of the standard organizational structures, i.e., functional, product, or matrix, that may be used for the project.

Size Element: The size of the effort required to develop or enhance a Mission Critical Defense System can influence the organizational structure chosen and used during the effort. During a small effort (up to 5,000 lines of code), an existing group of an established organization can assume and conduct management of the effort. A large effort (over 30,000 lines of code), may require establishing new organizations to manage the development or enhancement of the Mission Critical Defense System. The small effort can be supported by a product organizational structure or as an element of a matrix organization. However, a large effort normally has to be supported by a functional organizational structure, a matrix organizational structure, or a tailored combination of both the functional and matrix organizational structures. The rigidity of the Ada language and the inherent structured architecture support all sizes of development or enhancement efforts.

Type Element: The type of effort development or enhancement of a Mission Critical Defense System can also influence the organizational structure. The development type can be addressed by any of the organizational structures. The enhancement type normally occurs in a previously established organization using a functional organization or a combination matrix/functional organization.

Responsibilities Element: The current organizational responsibilities normally refers to an established functional organization or a combination matrix/functional organization. These are Life Cycle Engineering Centers or Mission Specific Organizations such as Fire Direction, Intelligence, Command and Control, etc.. Projected Mission Critical Defense System responsibilities deal with the proposed transition of a development system to a deployed system, which is to be enhanced and maintained, and expected to be supported by the organization types.

Organizational Structure Integration.

Once the structure has been chosen for the development or enhancement of the Mission Critical Defense System, the user's, customer's and contractor's organizational structures must be integrated to support the development and delivery of the product. Integration is normally accomplished by creating parallel organizations at the user, customer, and contractor levels, by identifying and establishing formal lines of communication, and by developing coordination responsibilities.

Creating Parallel Organizations: The project manager/customer organization is the organization that is normally paralleled by the contractor's organization. This paralleling leads to establishing technical points of contact at all three organizations which improve and speed the flow of technical information and the resolution of technical problems. The Project Manager's organization normally parallels the user's organization to improve and speed up the flow of requirements and development status information. The better the three organizations can parallel their structure, the more successful the integration and the coordinated effort to develop or enhance the Mission Critical Defense System.

Establishing Lines of Communication: The required lines of management communication are identified and tailored to fit the organizational structure selected for developing or enhancing the Mission Critical Defense System. Normally, direction flows down whereas information flows laterally and upward. The formal lines of communication are enhanced by the establishment of technical points of contact who are responsible for ensuring the flow of information and direction to the proper levels of the project. The complexity of the lines of formal communication is in direct relationship to the complexity of the organizational structure used for developing or enhancing the Mission Critical Defense System.

Developing Coordination Responsibilities: Coordination is a key element in the success of all projects requiring more than one organization to be responsible for completion of the tasks during development or enhancement of a Mission Critical Defense System. Coordinating the efforts of the user, the customer and the contractor during development or enhancement projects requires the development and implementation of a project plan. The project plan should define the work to be done and describe the plan for accomplishing that work. The management project plan should:

- Define the scope and use of the plan
- Identify controlling documents
- Provide project overviews, assumptions, deliverables, and schedules
- Provide detailed task descriptions, schedules, and budgets
- Describe project required support facilities, personnel, and services
- Provide a detailed description of the project development approach: techniques, services, documents, tools, etc.
- Describe project organizational structure and interfaces
- Describe standards to be applied to the project design, development, and test efforts
- Describe or reference Quality Assurance, Configuration Management, hardware/software integration, and security plans.

The first coordination task will be coordinating approval of the project plan. The same intensity applied to this effort should be applied to all future coordination efforts to ensure the successful development or enhancement of the Mission Critical Defense System. The responsibilities are identified in the plan.

Ada STRUCTURAL CONCEPTS

The Ada language is specifically designed to improve the quality of software by reducing programming and interfacing errors. Ada's strong data typing and numerous compile-time and run-time checks catch software problems other languages would not detect until or during integration testing. The Ada concepts to be used in developing or enhancing a Mission Critical Defense System should be identified to show how they are used in supporting Project Management elements to successfully complete the project.

Ada Concepts to be Applied.

The Ada concepts to be applied to the development and enhancements of Mission Critical Defense Systems can be specific to a life cycle phase, such as test and evaluation, or inclusive through multiple life cycle phases. The first Ada concepts to be addressed will be those that are limited to one or two life cycle phases.

Life Cycle Phase Specific Ada Concepts: The design phase of the Mission Critical Defense Systems project life cycle exhibits most of the phase specific Ada concepts being applied. Ada directly supports the use of metrics to monitor and evaluate computer resource utilization, software development and design personnel resources, the stability of requirements definition, and software design and development progress. These metrics evaluate such things as staffing, cost to date, machine utilization, module sizing, detecting faults, and status of software design and testing. Ada has an inherent capability for modeling which is used extensively in design. Ada's information hiding, data abstraction, packaging, modularity, and localization contribute to a more efficient and integrated system design. Software reusability is being incorporated into the design because of Ada's inherent support of the reusability concept. Development and testing phases are supported and improved by the use of Ada concepts in the design phase. Ada supports the use of automated Configuration Management tools for effective software and document change control and successful project task completion.

Life Cycle Phase Inclusive Ada Concepts: Probably the most comprehensive Ada life cycle phase inclusive concept is that Ada was designed or implemented with explicit support for software engineering built into the language. This support, in the form of rigidity and structured architecture that emphasizes modularity and localization, enables the creation of tools and environments that can be used to automate many aspects of large scale Mission Critical Defense Systems software development. Even without automation, the use of Ada improves the quality and productivity of the project engineering personnel in all phases of the life cycle. The Ada concepts in compilers and linkers used in software development, change, testing, and maintenance phases support configuration control for the systems which provides a stable baseline for controlling software and document changes. These concepts provide additional advantages in the development and maintenance phases by allowing separate compilation of packages.

Ada Concepts Supporting Project Management.

The Software Engineering explicit support designed into the Ada language exhibits the type of support provided to Project Management by Ada concepts. There are other specific concepts that provide Project Management support: Information hiding, data abstraction, modularity, localization, uniformity, completeness, confirmability, rigidity, structural architecture, and software reusability. Other Ada concepts, not included in this list, also contribute to integrating and supporting the Project Management effort for developing or enhancing Mission Critical Defense Systems. Project

Management support provided by Ada concepts can be viewed as direct, that which is visible/quantifiable, or indirect, that which is not visible in the process but visible in the resultant products.

Ada Concepts Directly Supporting Project Management: Ada concepts that are classified as directly supporting Project Management are those that enhance the support of the tasks performed regardless of the software language used. These are requirements definition, requirements analysis, system design, system development, verification and validation testing, acceptance testing, quality assurance, configuration management, and project control. The rigidity of the Ada language and the structured architecture enhance the support of Project Management in all these tasks by providing a structured environment. During the Mission Critical Defense System's life cycle, the Ada concepts of information hiding, data abstraction, modularity, localization, uniformity, completeness, and confirmability can be observed as individually and collectively enhancing the support of Program Management task accomplishment. Ada Compiler features such as date time stamping and separate compilation of packages support configuration management in the configuration control of the software. The metrics supported by the Ada language provide Program Management with the status and control information to perform their management functions. The results of applying these Ada concepts may not be immediately quantifiable, but they can be measured after the fact.

Ada Concepts Indirectly Supporting Project Management: The Ada concepts that are classified as indirectly supporting Project Management are those that enhance the support of the tasks performed during the development or enhancement of Mission Critical Defense Systems, but cannot be easily observed or measured until product delivery. The same Ada concepts that are classified as providing direct support also provide indirect support in the form of improving the quality of the Mission Critical Defense System software. The Ada concept of software reusability indirectly supports Project Management throughout the life cycle by incorporating and reusing verified and validated software, plus developing and maintaining a library of reusable software for developing or enhancing systems. The Ada concepts of uniformity, completeness, and confirmability indirectly support Project Management in evaluating and ensuring that these traits exist in the Mission Critical Defense System in addition to improving the quality of the product. The Ada concept of inherent modeling capability indirectly supports the Project Management task of evaluating and approving system design structure and progress.

MANAGEMENT APPROACHES TO Ada INTEGRATION

Current management approaches used by the Department of Defense in the development and enhancements of Mission Critical Defense Systems were not developed to maximize the use of Ada language concepts. These approaches and methodologies, which were designed, developed, and validated prior to Ada, use a functional approach. The new management approaches, such as Object Oriented Design, Jackson System Development, and TRW's Distributed Computing Design System, attempt to incorporate the Ada language concepts and features, but are limited in the portion of the Mission Critical Defense System Life Cycle which they address and have not been validated by the Department of Defense for system development or enhancement. The management approaches being used to integrate the Ada structures and constructs into the development of new systems and the enhancement of existing systems are not Ada concept maximizing approaches. This is an area currently being addressed, and the constantly improving approaches now being validated will address some of the existing management concerns to be resolved.

Management Approaches Being Used.

The approaches being used to develop and enhance Mission Critical Defense Systems are currently being fine tuned to address the inclusion of Ada concepts, but still utilize functional requirements and designs. Attempts have been made, on small Mission Critical Defense System developments, to develop requirements as functions then convert them to objects prior to design. This has met with limited success but does increase the use of Ada concepts during system design. The current approaches being used employ functional decomposition to design and develop the architecture for the software. The frame work employed in the approach is based on the reason for use and the inherent advantages of the frame work, i.e., waterfall, spiral, or Rapid Prototyping. The current methodologies being used, Structured Analysis and Design, Structured Development for Real-Time Systems, and Distributed Design vary the frame work but still rely on functional descriptions of requirement for design and development. The Ada concepts that can be identified and implemented in the selected approach are being incorporated and used to enhance the systems development and to support Project Management.

Existing Management Concerns.

New approaches for developing or enhancing Mission Critical Defense Systems need to support the integration of management levels as well as maximizing the use of the Ada concepts, constructs, structure, and architecture. Object Oriented Design needs to develop a front end process that develops user requirements as objects or converts functional requirements to objects. Major effort still needs to be applied to requirements definition. New methods should strive to incorporate graphical modeling techniques to represent the real-world behavior of the Mission Critical Defense System. Likewise, new methods should support the use of other formal specifications and establish limits for the use of Ada as a specification language. New approaches should incorporate the principals and concepts of objects and Ada representations in charts, diagrams, and program design language. The new approach selected must improve the integration of Ada concepts and management coordination during the development or enhancement of Mission Critical Defense Systems.

INCORPORATING Ada INTO MISSION CRITICAL DEFENSE SYSTEMS

Currently the incorporation of Ada into Mission Critical Defense Systems is accomplished either during system development or as an enhancement after the system has been developed and deployed during Post Deployment Software Support. New systems normally incorporate Ada partially or fully during system development. Existing systems can incorporate Ada either as an evolutionary development or as an enhancement during Post Deployment Software Support. Incorporation of Ada during development or enhancement has certain inherent benefits and associated risks which must be evaluated by Project Management before deciding when and how to incorporate Ada. Department of Defense progress in incorporating Ada in Mission Critical Defense System developments and enhancements reflects the current growth in use of the Ada language.

Incorporating Ada During Development.

Mission Critical Defense Systems development presents the opportunity for complete incorporation of Ada language concepts, constraints, architecture and software. Currently, there are two general approaches being used for the incorporation of Ada in new systems, either compete or partial incorporation of the language and its concepts. Each approach has its unique benefits and associated risks.

Compete Incorporation of Ada During Development: The complete incorporation of Ada during Mission Critical Defense System development includes not only the use of the Ada language but also the employment of related management concepts. Using current functional requirements development approaches does not maximize the use of Ada concepts whereas object oriented approaches have not been validated for major system development. The risk associated with developing a new Mission Critical Defense System in a new language is directly impacted by the development personnel's Ada experience level. Project Management must be prepared for the different emphasis and impact of Ada on the requirements definition, requirements analysis, software design, and software development phases of the Mission Critical Defense System that is visually uniform, structurally complete, logically correct, and provides the Ada benefits. The Ada benefits are reusable software, reduced programming and interface errors, improved software quality, and a more thoroughly tested Mission Critical Defense System. These are only a few of the factors to be considered by Project Management in evaluating when and how to incorporate Ada into a Mission Critical Defense System.

Partial Incorporation of Ada During Development. The partial incorporation of Ada during Mission Critical Defense System development may or may not include the employment of Ada related management concepts depending on the criticality of the portion incorporating Ada. Partial incorporation does not maximize the use of Ada concepts so functional requirements development approaches will require minimum or no modification. The risk of developing a new Mission Critical Defense System with a new language will be less and corresponding to the percentage of Ada code in the total system. The emphasis on Project Management during the life cycle phase will only be impacted by the percentage of Ada code in the total system. The benefits of using Ada will be limited and may not be observable. Again, these are only a few of the factors to be considered by Project Management in evaluating when and how to incorporate Ada into a Mission Critical Defense System.

Incorporating Ada During Post Deployment Software Support.

Post Deployment Software Support enhancement of an existing Mission Critical Defense System with Ada provides reduced risks to the system compared to complete incorporation of Ada in a new system. The risks associated with partial incorporation of Ada in development or enhancement are approximately equal and will be proportional to the percent of Ada incorporated in the Mission Critical Defense System.

Complete Incorporation of Ada During Post Deployment Software Support: The complete incorporation of Ada in a Mission Critical Defense System during Post Deployment Software Support would be an enhancement to the system. The complete incorporation of Ada would also include the use of the Ada language and the related management concept changes required during development. Post Deployment Software Support uses the same functional requirements development approaches as those used during a new development. The risk associated with the support contractor personnel's Ada experience level would be similar to that of a development contractor. Project Management will also have to modify the impacts and effects of Ada on the Mission Critical Defense System's life cycle. The benefits of a complete incorporation of Ada during Post Deployment Software Support are better coordinated requirements definition with the support contractor being closer to the user, and the greater ease of clarifying requirements when evaluating a product. The pressure to field a Mission Critical Defense System update is less than that for the initial fielding of the system. There are also lessor risks than those associated with developing a new system in a new language. The baselined environment of a Post Deployment Software Support Mission Critical Defense System provides stability to the enhancement process not available to a new systems development. These are only a few of the factors to be considered by Project Management in evaluating when and how to incorporate Ada into a Mission Critical Defense System.

Partial Incorporation of Ada During Post Deployment Software Support: The partial incorporation of Ada in a Mission Critical Defense System during Post Deployment Software Support enhancement may or may not require inclusion of Ada related management concepts, and will require minimum or no modification of functional requirements enhancement approaches. The risk associated with a new system in a new language does not exist, only the risk associated with a new language. The emphasis of project management on the Ada life cycle phase will also only be impacted by the percentage of Ada code in the total system. The benefits of using Ada will be more observable than in partial incorporation for a new system, due to the establishment and maintenance of a software reuse library in the Post Deployment Software Support Environment plus the retention and use of software portability information in this environment. As before, these are only a few of the factors to be considered by Project Management in evaluating when and how to incorporate Ada into a Mission Critical Defense System.

MISSION CRITICAL DEFENSE SYSTEM Ada PROGRESS

The progress being made by the Department of Defense in incorporating Ada in Mission Critical Defense Systems reflects on a limited scale the growth of Ada language use in industry.

THOMAS S. ARCHER

TELOS Systems Group
P. O. Box 909
Sierra Vista, Arizona 85636

THOMAS S. ARCHER has a Bachelors of Science in Mathematics. The author has over 29 years experience in the design, development, and implementation of complex Mission Critical Defense Systems application software. Mr. Archer is presently a Senior Systems Engineer for a Post Deployment Software Support contractor, Telos Corporation, providing technical support to the All Source Analysis System program and the system's proposed conversion to Ada. Mr. Archer has developed and presented "Ada in the Post Deployment Software Support Environment" at the October 1989 Tri Ada Conference and "Portability: A Key Element in Life Cycle Productivity" at the March 1990 Sig Ada Conference.

On Decomposing an Ada CSCI of a Large Command and Control System into TLCSCs, LLCSCs and Units: With Suggestions for Using DOD-STD-2167A

Lewis Gray

Ada PROS, Inc.

Abstract

This paper summarizes guidelines used to develop the Computer Software Components (TLCSCs, LLCSCs) and Units of a 30,000 line Computer Software Configuration Item (CSCI), coded in Ada, for a large U. S. Army command and control system, and discusses interesting results of their use. It concludes by offering suggestions for using DOD-STD-2167A.

The purposes of the paper are to present actual guidelines, and results of their use, and to alert projects to the pctential complexity of the decisions when CSCs and CSUs are created. An earlier version of the paper appeared in *Implementing the DOD-STD-2167 and DOD-STD-2167A Software Organizational Structure in Ada*, a report of the Association for Computing Machinery (ACM) Special Interest Group on Ada (SIGAda) Software Development Standards and Ada Working Group (SDSAWG) Software Organization Subgroup, August 1990.

1. Context

As part of the initial phase of a project to develop a large U. S. Army command and control system, object-oriented design (OOD) guidelines were developed and tested.[1]

Some of the guidelines were results of a study of the transition from structured analysis (for software requirements analysis) to object-oriented design. A review of the transitioning method developed by that study appears in an earlier paper.[2]

Other guidelines were results of an investigation of issues associated with developing the organizational structure of Computer Software Configuration Items (CSCIs), Top Level Computer Software Components (TLCSCs), Lower-Level Computer Software Components (LLCSCs) and Units that is required by DOD-STD-2167.[3] These guidelines are summarized in this paper.

This section of this paper first briefly presents the key concepts of the DOD-STD-2167 software organizational structure, for the benefit of readers who need this background (experienced users of DOD-STD-2167 or DOD-STD-2167A may wish to skip this discussion). Then it quickly summarizes the capabilities that CSCI designers had to provide, and the requirements that they were given.

Sections 2 through 5 of this paper discuss the most interesting guidelines and some of their rationales, reflect on their use during software development in 1987 (from Preliminary Design through CSC Integration and Testing), and offer suggestions on how to prepare to use DOD-STD-2167A.[4] The suggestions reflect three years of additional thought about the software organizational structure.

1.1 The DOD-STD-2167 "static structure".

Paragraph 4.2 of DOD-STD-2167, entitled "Computer software organization" requires software development contractors to organize their contractually-deliverable software into a hierarchical structure of CSCIs, TLCSCs, LLCSCs and Units. Figure 1 (Figure 3 of the standard, entitled "CSCI sample static structure") gives a graphical example of such a structure. CSCIs, TLCSCs, LLCSCs and Units can be called 'elements' of the structure.

The elements are defined in section 3 of the standard, then the definitions are extended and clarified elsewhere in the standard and in its Data Item Descriptions (DIDs). The discussion can be summarized as follows:

Computer Software Configuration Item (CSCI) - A CSCI is defined in terms of a configuration item, which is said to be "hardware or software, or an aggregation of both, which is designated by the contracting agency for configuration management."[5] The standard also says that a CSCI is a "part of a system, segment, or prime item."[6]

Computer Software Component (CSC) - "A functional or logically distinct part of a computer software configuration item. Computer software components may be top-level or lower-level [i.e TLCSCs or LLCSCs]."[7] "TLCSCs and LLCSCs are logical groupings."[8] They are also "architectural elements of the CSCI. . ."[9]

Unit - "Each Unit shall perform a single function."[10] A Unit is "the smallest logical entity specified in the detailed design which completely describes a single function in

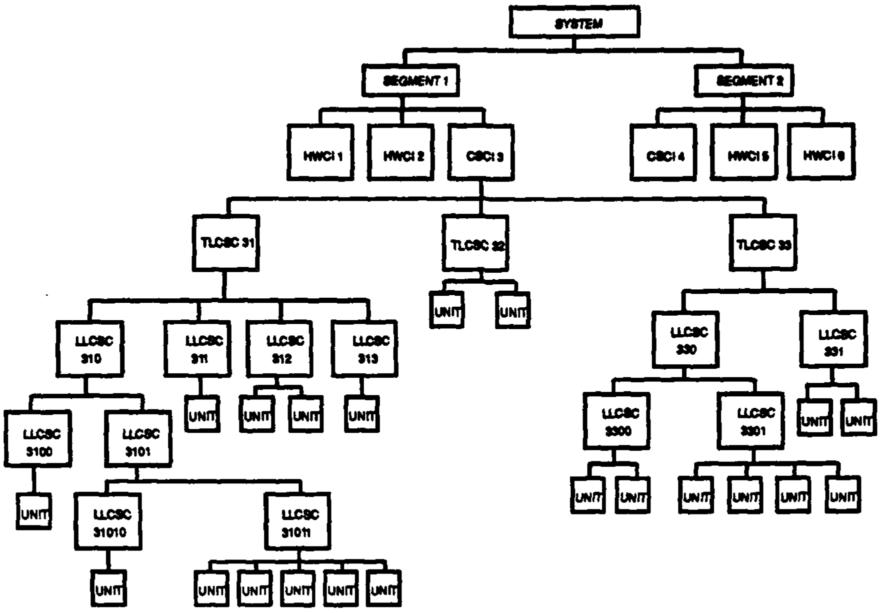


FIGURE 3. CSCI sample static structure.

Figure 1. The DOD-STD-2167 Software Organizational Structure

sufficient detail to allow implementing code to be produced and tested independently of other Units. Units are the actual physical entities implemented in code."[11] Like CSCs, Units are "architectural elements of the CSCI."[12] A Unit may be "a part of more than one TLCSC or LLCSC. . ."[13] And the standard suggests that they may be elements of more than one CSCI as well -- in particular a Unit may "reside in a library" for use "in many places" in which case it may appear in a design document suggested by the standard to describe another CSCI.[14]

The standard also indicates that CSCIs, CSCs and Units take part in several relationships. Five are specifically mentioned:

Consists Of - CSCIs "shall consist of one or more. . .TLCSCs. Each TLCSC shall consist of. . .LLCSCs or Units. LLCSCs may consist of other LLCSCs or Units."[15] In a similar passage, the standard refers to ". . .each CSCI and its constituent TLCSCs, LLCSCs and Units. . ."[16]

Decomposed Into - During Detailed Design, "each TLCSC is decomposed into a complete structure of LLCSCs and Units."[17] According to the standard, the decomposition is described in the SDDD.[18]

Make Up - ". . .various TLCSCs, LLCSCs and Units. . .make up the CSCI. . ."[19]

Partitioned Into - "The partitioning of the CSCI into TLCSCs, LLCSCs and Units may be based on. . ."[20]

Refined Into - During Detailed Design, the standard requires the contractor to "establish the complete, modular, lower-level design for each CSCI, by refining TLCSCs into LLCSCs and Units."[21]

Despite these definitions and clarifications, the standard never dictates exactly how the elements are to be associated with the contractually-deliverable software, that is, how CSCs and CSUs are to be mapped to things in an Ada program library. On every software development project, someone must decide. It is very important that this decision be carefully considered and well founded, because it can severely impact the project's cost and schedule.

1.2 The CSCI capabilities. Figure 2 summarizes the operational situation that CSCI designers faced. The basic requirements for the CSCI were to provide the capabilities for interactive users to store, retrieve and update information in several data bases, and to generate and transmit reports.

1.3 The starting point for software designers: Essential Systems Analysis. Essential systems analysis[22] is a refinement of structured analysis. Its standard products, similar to those of structured analysis, are Data Flow Diagrams (DFDs), a data dictionary that defines all the names (e.g. of processes, data flows, data stores) that appear on them, and mini-specifications for their lowest level ("primitive") processes. Tom DeMarco[23] calls these products, taken as a whole, a "system model" of the system of interest.

The starting point for software design was a system model contained in a Software Requirements Specification (SRS). There was a DFD in the SRS for each CSCI capability, that showed its decomposition into primitive processes.

The SRS for the CSCI was unusual in that all functional requirements in it were expressed by means of the system model. There were no "shall"-like statements of functional requirements, for example, except those in the system model. The paragraphs in the SRS where such statements would normally appear contained only pointers to the system model, which was attached as an Appendix to the document. All the CSCI's functional requirements were described there, either on DFDs or in their mini-specifications. In this SRS, it was common for a single functional requirement to correspond to half a page or more of text.

2. Guidelines for Decomposing the CSCI into TLCSCs, LLCSCs and Units.

Project guidelines for decomposing the CSCI into TLCSCs, LLCSCs, and Units were in three parts. First, there were guidelines for allocating requirements. These described how functional, interface and performance requirements in the CSCI's Software Requirements Specification (SRS) and Interface Requirements Specification (IRS) would be allocated to TLCSCs, LLCSCs and Units during Preliminary Design and Detailed Design. Second, there were guidelines for associating TLCSCs, LLCSCs and Units with parts of the CSCI's Ada program library.[24] Third, there were several guidelines for testing the composition of TLCSCs, LLCSCs and Units. In some cases, these were sophisticated "sanity checks" of the number and the roles of the elements. In other cases, they defined element documentation guidelines, including some that were met by means of the project's compilable Ada Design Language (ADL) standard.

2.1 Allocating functional, interface and performance requirements. Creating CSCs and Units can require complex decisions and creative thought equivalent to that required for software design and functional area management. It must be guided by an understanding of the software because it will determine how the software's design will be presented during the formal design reviews (Preliminary Design Review (PDR), Critical Design Review (CDR)). It must be guided also by an understanding of management and technical tasks such as calculating design progress, informal testing, and controlling the CSCI's developmental configuration. The way the elements are created can make each of these tasks significantly harder or easier than expected. For example, a well-known design progress metric is the percentage of Units that have been completely designed.[25] This metric is reasonable only for the case where the Units are more-or-less equivalent in size and complexity. Otherwise the simple, unweighted ratio should be altered to take account of their varying difficulty, since it will affect their cost to complete.

Specifically to ensure that impacts on software project management and the other functional areas of the project would be considered, responsibility for creating the TLCSCs was assigned to a person with an understanding of the entire development effort. The software design team leader during Preliminary Design, who was responsible for performing the initial decomposition of the CSCI into TLCSCs, was responsible (de facto) for establishing cooperative working agreements among most of the major functional areas of the project, i.e. software engineering, formal qualification testing, software product evaluation, and software configuration management.

Sample guidelines for allocating requirements to software organizational structure elements are listed in Table 1.

2.2 Associating CSCIs, TLCSCs, LLCSCs and Units with subsets of Ada program libraries. Earlier investigation of the software organizational structure in 1986[26] led the project to implement CSCIs, TLCSCs, LLCSCs and Units in terms of sets of Ada compilation units in the following way:

CSCI — A set of compilation units that is also a configuration item. CSCIs contain CSCs and Units (as subsets). The CSCI for the project was described by a Software Requirements Specification (SRS), an Interface Requirements Specification (IRS) and a preliminary Database Design Document (DBDD).

CSC (TLCSC) (LLCSC) — A set of compilation units. CSCs contain Units (as subsets). Although changes to CSCs are controlled during the development cycle, CSCs are not

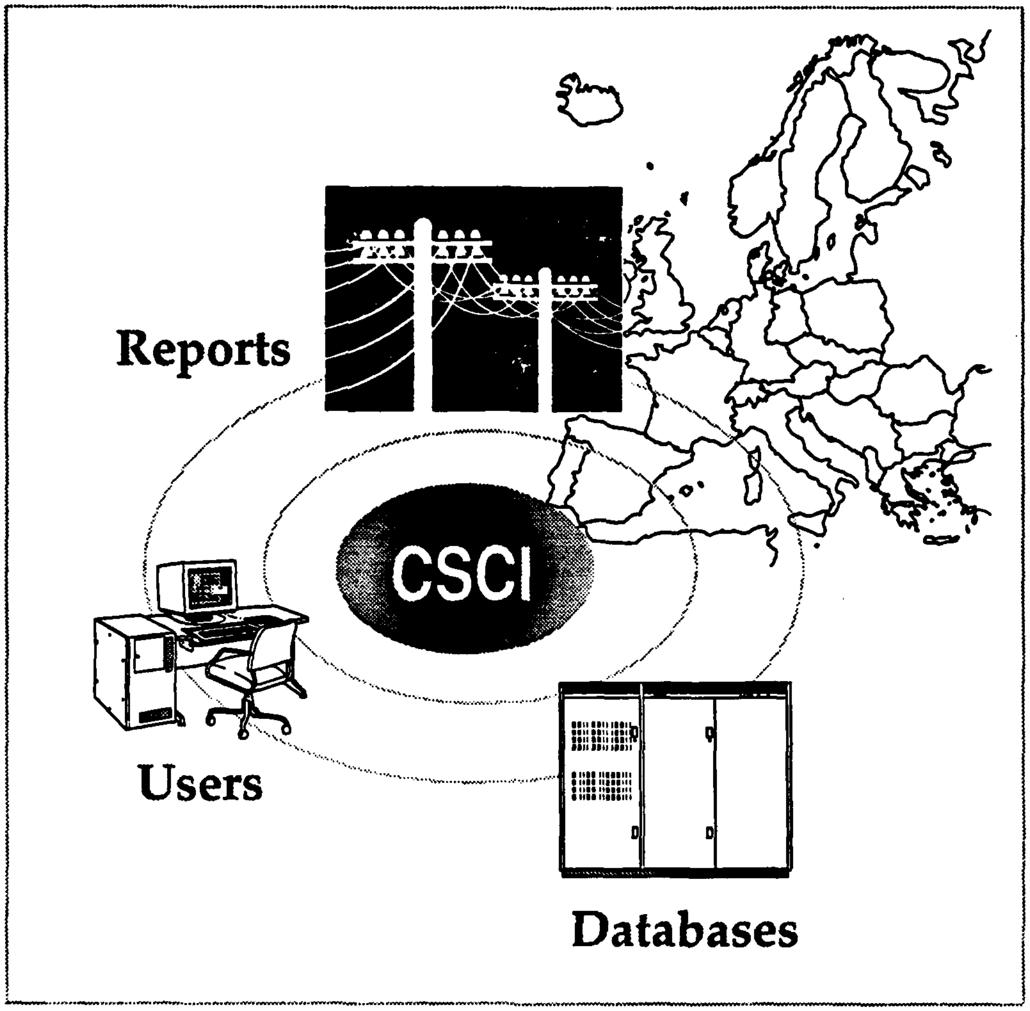


Figure 2. CSCI Capabilities

configuration items. Typically, a CSC will satisfy several functional, interface or performance requirements that are logically cohesive from the point of view of the customer, or the eventual CSCI users.

Unit One or more Ada compilation units that define a library unit, and its associated secondary units. ANSI/MIL-STD-1815A-1983 defines a library unit as a subprogram declaration, a package declaration, a generic declaration, a generic instantiation, or a subprogram body.[27] The secondary units can be described as a compilation unit that defines the proper body of a library unit, or a compilation unit that defines the body of a program unit that is declared within a secondary unit.

Figure 3 graphically summarizes this approach down to the level of the first subunit.

Project design and coding standards led designers to compile program unit specifications and bodies separately. This resulted in sets of physical files for the CSCIs, CSCs and Units that were suitable for organizing in a structure of file directories that mirrored the software organizational structure. One influence to take this approach was the DOD-STD-2167 requirement that Units must be testable independently of other Units. The project recognized early that this could be accomplished elegantly by an implementation strategy that made it possible to simply enclose each Unit in test drivers for Unit testing.

2.3 Testing the Composition of TLCSCs, LLCSCs and Units.

2.3.1 "Sanity checks". Table 2 lists sample guidelines for checking the reasonableness of the choice of elements. The purpose of creating such a list was to provide a tool for checking the quality of the software organizational structure before production of the Software Top Level Design Document (STLDD) began. We reasoned that correcting a poor element early in design would be much less expensive than correcting it after delivery of a design document.

Some of the guidelines in Table 2 are adaptations for CSCs and Units of criteria for selecting configuration items that appear in Appendix XVII of MIL-STD-483A.[29] DOD-STD-2167 suggests in paragraph 4.2.1 that they may be used for creating CSCs and Units as well. The project resolved to test that suggestion.

Table 1. Some Interesting Guidelines for Allocating Requirements to *TLCSCs, LLCSCs* and *Units*

1. CSCs will be allocated one or more functional, interface or performance requirements.
2. The initial allocation of requirements to TLCSCs will occur prior to the development of the Ada program library units that will eventually constitute them. The allocation process is a device for subdividing the design work among multiple design teams.
3. At the time CSCs are first defined, functional similarities among requirements in the requirements database, suggested by mini-specs and the initial, candidate list of classes, objects and operations,[28] will guide the creation of an appropriate number of functional groupings of requirements. Each of those functional groupings will be allocated to a TLCSC.
4. Functional requirements which are highly data or control interdependent should be allocated to the same CSCs. Functional requirements which exhibit a high disparity between input and output data rates should be allocated to separate CSCs.
5. TLCSCs and LLCSCs will be assigned to a "team" of software designers. A team could consist of a single designer, but more often it will probably contain 2-3 designers. Units will be assigned to individuals.
6. Units will be allocated only a single functional or interface requirement.
7. Other things being equal, one allocation of requirements to CSCs and Units will be preferred over another if it is clearer in the first case how requirements in the requirements documents have been grouped.

2.3.2 Element documentation guidelines. Project guidelines called for developers to provide the information shown in Table 3 for each Ada program unit developed in the project's compilable Ada Design Language (ADL). Much of the information exists to respond to requirements in the Data Item Descriptions (DIDs) for the deliverable design documents. But some of it was suggested by an early paper on IEEE Project 1016, which presented recommended practices for describing software designs.[30]

Table 3 describes project ADL guidelines for documenting Ada program units. Where are the guidelines for documenting elements of the software organizational structure? By associating the elements with Ada program libraries, as described in paragraph 2.2 of this paper, the project was able to combine the documentary information about Ada program units into documentation of the elements, and thereby provide much of the information about the elements that was required by the Software Top Level Design Document (STLDD) and Software Detailed Design Document (SDDD) DIDs.

3. Rationale.

The guidelines for decomposing the CSCI resulted from investigating fifteen decomposition considerations.[31] Several of the considerations are listed in Table 4. They fall into the following five categories:

1. Descriptions of the software organizational structure elements in the applicable standard (DOD-STD-2167 and its DIDs);
2. Criteria for selecting configuration items in MIL-STD-483A;
3. Software design vs. design presentations;
4. Presenting a design using Ada as a design language; and
5. Expected users of the software design documents.

The goal of the investigations was to review every obvious potential influence on decomposition criteria before choosing decomposition guidelines for the project. Each of the categories is a major source of influence on how contractors should choose the information to present at the formal design reviews

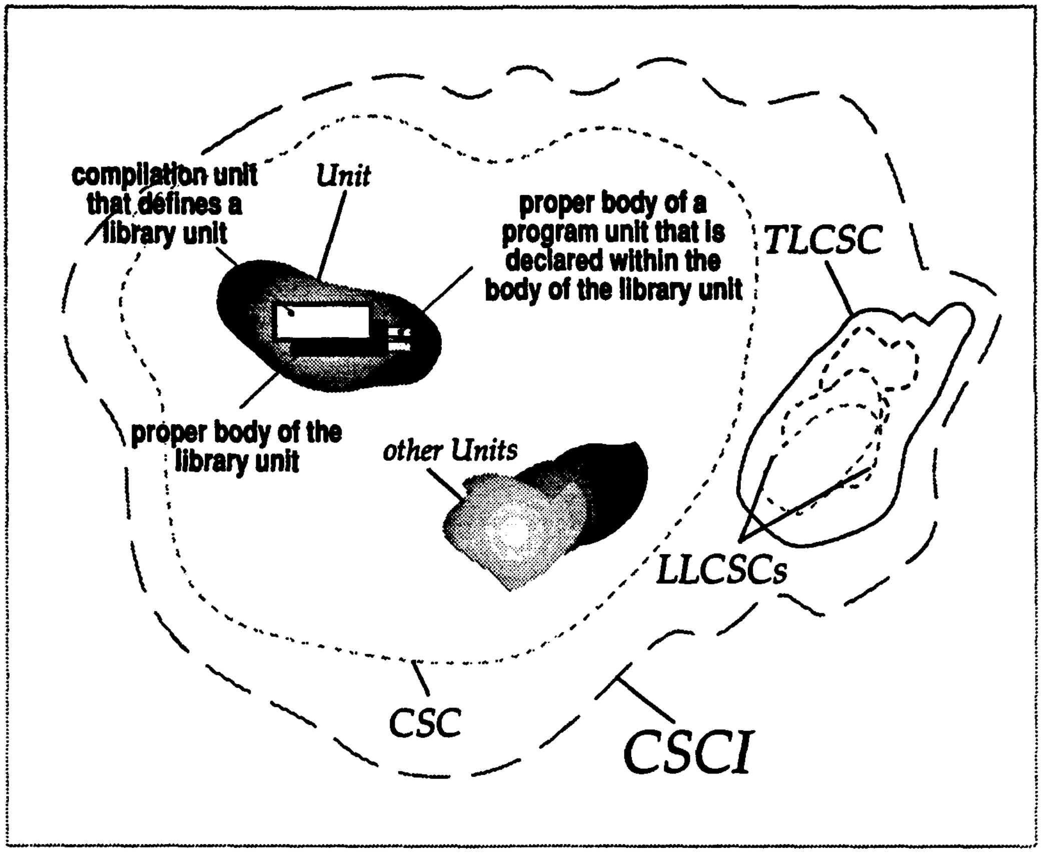

compilation unit that defines a library unit
Unit
proper body of a program unit that is declared within the body of the library unit
TLCSC
proper body of the library unit
other Units
LLCSCs
CSC
CSCI

Figure 3. Associating TLCSCs, LLCSCs and Units with Subsets of an Ada Program Library

(PDR, CDR). The number and the complexity of these influences are normally not discussed in papers about implementing CSCIs, CSCs and Units/CSUs. Projects should bite the bullet and consider all of them, at least once, before they start down the long, long road of a DOD-STD-2167 or DOD-STD-2167A software development contract.

To give the reader a feel for what was done by this project, four of the considerations are reviewed below.

3.1 When implementing the static structure, logical entities should replace logical elements, and physical entities should replace physical elements.

DISCUSSION: DOD-STD-2167 explicitly introduced a new distinction between logical things and physical things. One of the logical things that it mentions is a "logical grouping". Although the standard does not mention it, a very familiar example of a logical grouping is a set. The example that the standard gives of a physical thing is code. The point of this consideration is that things that the standard calls logical, but not physical, CSCs, should not be implemented by things that the standard calls physical, code.

Some argue that an Ada package that collects logically related types or operations is a logical grouping. But they also have to admit that it is code -- there is no question about that. And the standard calls code physical. So the package, or any other program unit for that matter, was not felt to be a good implementation of a DOD-STD-2167 CSC.

Because DOD-STD-2167A dropped the terms 'logical' and 'physical',[32] this constraint does not apply to CSCs described by the new standard.

3.2 CSCs and Units should be chosen by managers at some level of the project, to establish the "optimum management level" for the project, i.e. the level below which they delegate control, and give up visibility in the period between reviews.

DISCUSSION: DOD-STD-2167 states that guidelines for selecting CSCIs contained in MIL-STD-483A, Appendix

Table 2. Some Guidelines for Checking the Reasonableness of Software Organizational Structure Elements

1. No requirement will be allocated to a CSC or Unit that does not appear in a requirements document.
2. No requirement will be allocated to a LLCSC or Unit unless it is also a requirement of at least one TLCSC that encloses it.
3. Functional requirements allocated to a CSC or Unit should not be partitionable into subsets that are local to separate geographic areas. Moreover, requirements allocated to physically distinct processors in a distributed environment should be allocated to separate CSCs or Units as well.
4. Compilation units provided by different suppliers should be assigned to separate CSCs or Units.
5. A collection of compilation units should be identified as a separate CSC or Unit if [its failure] would adversely affect security, human safety, the accomplishment of a mission, or nuclear safety, or would have a significant financial impact.

XVII, "may also be applied to selecting TLCSCs, LLCSCs and Units."[33] MIL-STD-483A describes CSCIs as management tools. It calls their selection "a management decision"[34], not a technical decision. The selection "reflects an optimum management level during acquisition. . .at which the contracting agency specifies, contracts for, and accepts individual elements of a system."[35] Management below a certain level of detail is known to be counterproductive. So, "the selection is normally limited to the designation of configuration items to major subsystem levels of the Work Breakdown Structure, or to a critical item of a lower level. . ."[36] But the standard cautions that "choosing too few or the wrong elements as configuration items runs the risk of too little control through lack of management visibility."[37] If CSCIs are management tools, then CSCs and Units must be as well.

The qualifications of the managers considered here were assumed to include a position sufficiently high in the project's organization that they were familiar with all its CSCIs and met regularly with peers in all the functional areas of the project, for example software development management, software engineering, formal qualification testing, software product evaluation and software configuration management. The managers were assumed also to be sufficiently close to the software engineering technical decisions, and sufficiently experienced technically, that they were capable of constructively contributing to the decisions, technically.

3.3 Neither the elements of the static structure, nor the relationships between them, need reflect the methods used by designers to solve the problems of 1) choosing data and control structures for the software, or 2) decomposing the software into modules.

DISCUSSION: David Parnas argues that "those who read the software documentation want to understand the programs, not to relive their discovery."[39] He compares the process of designing and documenting software to discovering and publishing mathematical proofs:

> "Mathematicians diligently polish their proofs, usually presenting a proof very different from the first one that they discovered. A first proof is often the result of a tortured discovery process. As mathematicians work on proofs, understanding grows and simplifications are found. . .The simpler proofs are published because the readers are interested in the truth of the theorem, not the process of discovering it."[40]

Parnas explicitly recommends that projects should distinguish between the design activity where one discovers what data types, operations and modules are appropriate for implementing a system, and the presentation activity where the results are explained. An interesting paragraph in MIL-STD-881A makes the same point in a different way: "all reporting requirements for the project shall be consistent with the project/contract [Work Breakdown Structure] WBS. The organization of reporting requirements shall not be construed by either the DoD component or the contractor as determining the manner in which the defense materiel item is to be designed or produced."[41] In this case what is true for a missile or a ship is true for a CSCI also: it is a mistake to confuse the way a contractor describes progress with the way a product is designed or produced.

The discovery process and its methods, for example object oriented design, are not the topics of software design documents, any more than the means by which a mathematical theorem was discovered is the topic of the proof, unless the designers have to fall back on certain principles of the design method in order to explain the design.

Table 3. Some of the Information to be Provided About Each ADL Program Unit

1. The origin of the software [e.g. another vendor, internal development].
2. If developed internally, the names of its authors.
3. An explanation of why it exists, i.e. what role it plays in the design. This can include a description of the control logic that governs invocations of its compilation units, e.g. the normal timing and sequencing conditions under which they execute.
4. A statement of what it does, i.e. what requirements in the requirements specifications it satisfies.
5. A description of how external devices are used by it.
6. A description of the algorithms carried out by it.
7. A list of exceptions that may be raised within it, and the conditions under which they may be raised.
8. A list of exceptions that may be handled within it.
9. The major information sources (references) used to suggest or guide its design.

3.4 Expressing a design in a compilable Ada Design Language is implementation ("coding").

DISCUSSION: To avoid semantic clashes with Government reviewers about whether its compiled ADL was source code, the project stated its position at the start that it was. Two simple tests indicate that this is the case. First, since the design language is compilable by an Ada compiler, it satisfies the definition of the Ada language. Thus the product of writing in the design language is Ada source code, regardless of how it is advertised. Second, if the design language text constitutes a necessary part of the deliverable program libraries for the system, then it is a part of the system's source code by definition -- that is the system would not function properly if it were removed. In that case, the act of writing it is obviously "coding", or implementation, as those terms are used in every day speech, regardless of when that act occurs in the life cycle.

Other related considerations that are not discussed in this paper (for example 6 and 7 in Table 4) attempted to test whether it was practically possible to identify a design subset of the deliverable source code that was appropriate for presentation at formal design reviews (Preliminary Design Review (PDR) and Critical Design Review (CDR)).

The driving question behind all the project's work on these topics was, exactly what information about the software design should the project present to the Government at PDR and CDR? David S. Maibor, the principal author of DOD-STD-2167, and others[42] emphasize how important the question is. It is only part of a larger question that goes to the core of why the Government releases Data Item Descriptions (DIDs) with DOD-STD-2167 and DOD-STD-2167A: what does a Government Program Manager (PM) need to know in order to effectively manage a large software development contract?

4. Reflections on the Use of the Guidelines.

Most of the guidelines listed above were carried out. Some of the most interesting results of using them are described below.

4.1 The project's review contractors were out of step with evolving Government thinking about the software organizational structure. The project's definition of a Unit as a collection of compilation units belonging to the same Ada program unit, and a CSC as a set of Units, is echoed in a recent Government paper by the Joint Logistics Commanders Subgroup on Computer Software Management (JLC/CSM).[43] The project's review contractors in 1987 attacked the definition because it defined a Unit solely in terms of Ada program library units. The reviewers advised the Government contracting activity that, at a minimum, the definitions should add the requirement that Units be Ada packages, and should forbid the use of Ada procedures and functions as library units, except in unusual cases. The Government paper by the JLC/CSM three years later sides with the project on this issue.

In another example of the same kind of situation, review contractors at the time heavily criticized the project's decision to reject all requirements that relationships between the software organizational structure elements, i.e. the lines between elements in Figure 3 in DOD-STD-2167, must reflect control or data flows, or visibility / compilation relationships, or program unit nesting within the deliverable software. Although the project's position was consistent with paragraph 4.2.1 of the standard, and it has turned out to be the same position that the Government takes in the recent JLC/CSM paper, reviewers advised the customer in 1987 that the software organizational structure should correspond to the compilation dependencies among Ada compilation units.

4.2 Bias in favor of object-oriented techniques prevented the creation of functional CSCs. Guideline 3 in Table 1 was never followed. External objections by reviewers who felt that object-oriented software design should result in an object-oriented software organizational structure, and internal objections by some project staff members with the same idea, made it impractical to carry it out. As a result, the CSCs were created to collect Ada program units that implemented similar objects. For example, objects that represented databases were collected together in the Data Base Interfaces TLCSC.

Table 4. Several Interesting Considerations that were Investigated

1. When implementing the static structure, logical entities should replace logical elements, and physical entities should replace physical elements.

2. The lines on the static structure should not be interpreted as a representation of the control flow within the CSCI, or of nesting within the implemented code.

3. CSCs and Units should be chosen by managers at some level of the project, to establish the "optimum management level" for the project, i.e. the level below which they delegate control, and give up visibility in the period between reviews.

4. Neither the elements of the static structure, nor the relationships between them, need reflect the methods used by designers to solve the problems of 1) choosing data and control structures for the software, or 2) decomposing the software into modules.

5. Expressing a design in a compilable Ada Design Language is implementation ("coding").

6. Writing program unit specifications is implementation, and may be design as well. Writing program unit bodies is only implementation.

7. Every main program in the system presents design information, and they should all be described in the design documents for the system. The specification of every program unit that is used by the main program (that is named in a "with" clause[38]) should also be described somewhere in the design documents, and so should the specifications of all the program units that are used by their specifications, and so on in a transitive manner.

This approach caused a serious problem with requirements traceability matrixes in the software design documents. It led also to the need to create functional threads for CSC Integration and Testing. Both of these impacts are described below.

4.3 Satisfying functional requirements with object-oriented CSCs led to uninformative requirements traceability matrixes in the software design documents. The functional requirements for the CSCI, written into its SRS, were components of data flow diagrams (DFDs), primarily their mini-specifications, i.e. the processing logic of the DFD's lowest-level primitive processes. Most of the requirements included at least one operation on a database. Since all objects that represented databases were collected into the Data Base Interfaces TLCSC, most of the functional requirements traced to that CSC.

The situation was similar, or more exaggerated, for most of the other CSCs. For example, there was an Events TLCSC that, by definition, collected objects associated with every functional requirement. As a result, all functional requirements traced to it. The impact on the requirements traceability matrix in the STLDD, for example, was to link nearly every TLCSC to each functional requirement.

This sent the wrong message to the Government that to verify that a single functional requirement had been satisfied by the design, the Government would have to examine all the compilable Ada Design Language (ADL) for the entire CSCI. The real situation was that only a limited amount of the design contributed to satisfying any single functional requirement. This became clear much later during CSC Integration and Testing. But, because of the particular combination of the way requirements were represented in the SRS and the way CSCs were created, the requirements traceability matrixes in the software design documents hid this fact.

4.4 Although the CSCs were object-oriented, they had to be tested by means of functional threads. DOD-STD-2167 only requires (paragraph 5.5 and its subparagraphs) that contractors test aggregates of Units during CSC Integration and Testing. It does not require that the aggregates be identical to CSCs. It became apparent before the CSC Integration and Testing phase that only functional aggregates of Units could be informally tested against the functional requirements in the SRS. So, the project defined functional threads, one thread per functional requirement, that consisted of all the Units needed to satisfy that requirement. The threads crossed all the TLCSCs of the CSCI. They were documented in a special set of thread development files, similar to the Software Development Files (SDFs) for the Units.

4.5 A small change in the guidelines could have led to doubling the size of the SDDD, despite no change at all to the deliverable software. The guidelines in paragraph 2.2 above resulted from a study of the software organizational structure that began in 1986.[44] They were carefully matched to the project's design and coding standards to provide the kind of information appropriate for the formal design reviews (PDR, CDR) without causing excessive documentation.

Alternate guidelines in use by another Ada project in the same company could have been used. Although project personnel were not aware of this at first, they could have led to a substantial increase in the size of the software design documents. The alternate guidelines were not used by the project, but for a different reason because they did not seem as good at achieving the structure of logical and physical elements that DOD-STD-2167 required. However, while evaluating the project's guidelines after CSCI Testing, it was discovered that the alternate guidelines could have increased the number of CSCs in the CSCI from 36 to 229 TLCSCs plus an unknown number of LLCSCs, more than six times their original number. They could have increased the number of Units from 229 to 845, more than three times their original number. Yet there would have been no change at all in the software itself. Estimating conservatively, the size of the SDDD (for the same software) easily could have doubled if the alternate guidelines had been used.

4.6 The use of object-oriented design led to a kind of internal software reuse that violated the guideline to assign only a single functional requirement to a Unit. Guideline 6 in Table 1 states that a Unit could trace back to only a single functional requirement. The guideline was an attempt to comply with what seemed to be the spirit of the definition of Unit in paragraph 3.23 of DOD-STD-2167, and the requirement in paragraph 5.3.1.2 of the standard that "each Unit shall perform a single function."

It happened that functional requirements for the CSCI were equivalent usually to several "shall" statements, because each functional requirement was the processing logic of a Data Flow Diagram (DFD). Because many of the functional requirements in the SRS were similar to one another in the general nature of what they described, for example interactive use of a database, an object developed for one requirement was often used for many, which violated the guideline.

4.7 Too much project time was spent on internal disputes about the guidelines. There are topics, like security, contracts, accounting, database design, testing and Ada(!!!) where people seem reluctant to take a strong personal stand that contradicts what specialists say. This is not the case for the elements of the software organizational structure. The guidelines described in this paper were carefully explained to every one on the project who showed an interest. Whether because of this openness or not, people seemed surprisingly disposed to challenge the guidelines with personal positions which they defended aggressively and doggedly. One example of this has already been mentioned above, the position that object-oriented software design should result in an object-oriented software organizational structure. Another example is the belief held tenaciously by some at the time that DOD-STD-2167 requires contractors to integrate Units into CSCs, test the CSCs in isolation, and then integrate the CSCs into a CSCI. In fact, paragraph 5.5 of the standard and its subparagraphs only require that aggregates of Units be tested. The standard leaves it to the contractor to choose which Units to aggregate for testing.

Over time, positions like these burned up significant project resources in inconclusive meetings and continual debates.

4.8 The guidelines freed key software designers from worrying about how to comply with DOD-STD-2167 documentation requirements, which let them concentrate on solving software design problems. The guidelines for decomposing and documenting CSCIs, together with the project's Ada software design and coding standards and procedures, were in place before software design began, and they addressed all of the DOD-STD-2167 documentation requirements. The combination of the guidelines and the design and coding standards and procedures specified what design characteristics and decisions were to be documented where. Much of the documentation was done within the Ada design language for the CSCI, as a natural part of design, through structured comments like "--| F: 101.1.1.4.19 Update_EPW_Evacuation" which marks a compilation unit that satisfies functional requirement 101.1.1.4.19 in the SRS. Key designers concentrated on applying the project's chosen object-oriented design method to satisfying the requirements in the SRS.

A few decisions about compliance with DOD-STD-2167 design document DIDs still remained. Many of these addressed the content of the charts and diagrams to include in the design documents, for example data flow or control flow diagrams, compilation dependency diagrams, call-tree diagrams showing which program units invoked which others, performance charts (sizing and timing characteristics), and requirements traceability matrixes. A major decision had to be made about how to group the Ada program units that the designers were creating into TLCSCs, and LLCSCs. All of these remaining decisions about compliance with the standard were assigned to the very small group of top design leaders who were software designers themselves but with overriding responsibilities for meeting schedules and satisfying both technical and contractual requirements, and who had frequent contact with their peers in functional areas like configuration management and testing, outside their own area of software engineering.

5. Suggestions for Using DOD-STD-2167A with Ada.

Several important lessons glimmer among the results described above.

5.1 Don't assume that software designers are the best qualified to create CSCs and CSUs, because a CSCI's software organizational structure is not its software architecture. It is clear that it is possible to change the CSCs and Units of a CSCI without changing its software. For example, the number of TLCSCs and Units for the CSCI could have been increased dramatically merely by changing the guidelines in paragraph 2.2 above. The reason for this is that individual CSCIs, CSCs and Units just named collections of software. For example, Data Base Interfaces, one of the CSCI's TLCSCs, was just a name for a collection of Ada compilation units that implemented objects of a particular kind. It is easy to see that a different collection could have been created without changing the software itself, just by drawing the boundaries of the CSC in a different way. For another example, although the CSCs for this project were object-oriented, when the time arrived for CSC Integration and Testing, they were tested in functional threads. The guidelines above would have allowed us to simply call the functional threads CSCs in the first place, and this would not have affected the software in any way.

The software architecture of a CSCI is a high-level view of its software design. It is an abstraction of the software that shows the relationships among its significant parts. To change the software architecture of the CSCI, it is necessary to make a substantive change to some characteristic of the software, like its compilation dependencies, its calling relationships, its algorithms or the data structures that it creates. One has to change a part, just like to change a compilation dependency in an Ada program, one has to change a "with" clause, or create a subunit. Merely giving a new name to a collection of compilation units does not get the job done.

Since we can change the software organizational structure without changing the software architecture, and vice versa, it is clear that they are not the same thing, and that the software organizational structure is not the software design either. It follows from this that software designers are not necessarily any better at creating CSCs and CSUs than anyone else.

5.2 Expect that confusion and debate about the software organizational structure will complicate software development contracts for years. David Maibor often says about Figure 1 above that "illustrating a CSCI's elements with this tree structure has caused countless problems for the Government and industry."[45] In support of this, approximately 40 per cent of the issues in the JLC/CSM paper deal, in one way or another, with the nature of CSCs and CSUs or how they should be implemented in Ada. Similar issues appeared in the 1986 SDSAWG *Issues and Subissues Report*.[46] The longevity of the issues suggests their difficulty. Also, the prolonged disagreements about guidelines that are described above suggests that people tend not to change their positions easily on these topics.

This situation is part of a larger problem. In a presentation to the Department of Defense Software Working Group, in support of the DoD Software Master Plan, Ole Golubjatnikov stated that "the main cause of cost escalation in defense software development is the improper application, interpretation, demonstration of compliance, and enforcement of specifications and standards."[47] He went on to say what many feel in the defense software industry, that many organizations, both in industry and Government, are still far from understanding how to use DOD-STD-2167 or DOD-STD-2167A effectively.

Papers like the JLC/CSM paper help to counter the problem. The Government has started to extend the tailoring handbook for DOD-STD-2167A[48] to include an additional Rationale volume and an Application volume. If these are completed and released, they will help much more. But I believe that years will pass before the problem is solved.

Meanwhile, software development contractors, their customers and the customer's reviewers often disagree about what the elements of the software organizational structure are. Meyer et al.[49] describe a recent case like this.

5.3 So long as controversy and confusion persist about how to develop the software organizational structure, delegate the job to specialists who have technical and management-level understanding of the project's deliverable data items, and regular access to the customer. 4.3 through 4.5, and 4.7 above hint at the potential cost of misunderstanding the software organizational structure. A personal, non-scientific poll of colleagues over the past few years has persuaded me that development technicians in several functional areas of large software development contracts waste half of their working hours during software design trying to understand DOD-STD-2167A so that they can comply with it when the time comes to deliver data items to their customers. The immediate result is that scheduled technical activities languish during this period. Moreover, if a contractor's understanding of DOD-STD-2167A leads it into a conflict with its customer, or its customer's reviewers, which often happens, responding to requests for clarification and deficiency reports from the Government can burn up a significant part of the remaining budget for design. The ultimate result is that the software is completed later than scheduled, at a higher cost than budgeted.

It is common practice for contractors to hand the DID for the DOD-STD-2167A Software Design Document (SDD) to a software designer and say, in effect, go figure it out and write the document. Typically, this practice just adds another untrained opinion to the on-going debate about what CSCs and CSUs are. Often, Ada software designers do not have enough job experience to understand project management, software configuration management and informal testing. Also, they are not familiar with the situation of Government employees or contractors who have to review several thousand pages of deliverable documentation in one or two weeks. Like the other technical personnel on the project, they are also affected by compartmentalization of development activities along the lines of the major functional areas in DOD-STD-2167A, i.e. software development management, software engineering, formal qualification testing, software product evaluation and software configuration management. Typically, it is expected that they will understand their part of the software development process very well, but only vaguely understand the rest.

A better way to prepare an SDD would be to create a team consisting of the software designers and a seasoned DOD-STD-2167A document development specialist with an Ada software development and project management background. The designers would design, using the tools and methods that were best suited to the application domain and target hardware, and the post-deployment software support conditions (who takes the time to review these now?). The specialist would develop the SDD, consistent with the other applicable deliverable documents, for example the SRS and IRS from the earlier Software Requirements Analysis phase, and the Interface Design Document (IDD), the Software Test Plan (STP), the Software Test Description (STD), and the Software Development Files (SDFs) that are created at the same time as the SDD, and deliverable documents required by other standards that might have been contractually imposed such as DOD-STD-7935A[50] or DOD-STD-1467 (AR)[51]. The specialist would guide compliance with the software configuration management procedures in the Software Development Plan (SDP), and assist configuration management and quality audits. Finally, the specialist would maintain contact with the customer for the purpose of keeping up-to-date on what to tell the customer about the design at the formal design reviews.

Think of the specialist as an architect who builds the various and different data items on the Contract Data Requirements List (CDRL) into a single, integrated logical work product. What do I mean by "logical work product"? I mean that the data items would be as consistent and complementary and non-redundant when they were delivered as they would have been if they had been developed by a single person. For any given fundamental (or "atomic") piece of information required by a DID, it is possible for a project to know whether it is required by the DID of another CDRL item and who its authors will be in each case. When this is known, the project can require that one author write it, and the other authors reference it, thereby achieving consistency across documents and reducing cost. Software designers are not interested in such work, and they have no time to pursue it in most cases even if they were.

The product architects would report to a product architecture group to be managed by a second-level project manager. Figure 4 shows the product architecture group on a sample organization chart for a large project.

5.4 Establish a product architecture group now. First, software contractors who expect to obtain large Ada software development contracts from the DoD should establish immediately a permanent product architecture group staffed by specialists on developing Ada software in compliance with DOD-STD-2167A. This group should begin immediately to analyze the DIDs associated with the standard and with other standards that might be imposed on the same contracts. The goal of this work should be to identify the atoms of information that are assembled in different ways into all the deliverable data items for the expected contracts. The group should also produce a plan for reusing the information during the course of a contract.

Second, prior to contract award, the product architecture group should deliver to the customer a proposed tailoring of the applicable standards and associated DIDs for that contract.

Third, following contract award, the product architecture group should coordinate development of all CDRL items. It should establish a system for reusing the information in the data items that complies with the reuse plan. And it should define the configuration items in the System / Segment Design Document (SSDD), and the software organizational structure elements in the SDD.

[1] *OOD Readiness Task Report, Volume I - Procedures for Object Oriented Design*, and *Volume II - Rationale*, Army WWMCCS Information System (AWIS), Project Management Office, Fort Belvoir, VA, 22 May 1987.

[2] Gray, Lewis, "Transitioning From Structured Analysis to Object-Oriented Design," in *Proceedings of the Fifth Washington Ada Symposium*, June 27-30, 1988, pages 151 - 162.

[3] DOD-STD-2167, *Defense System Software Development*, U. S. Department of Defense, 4 June 1985.

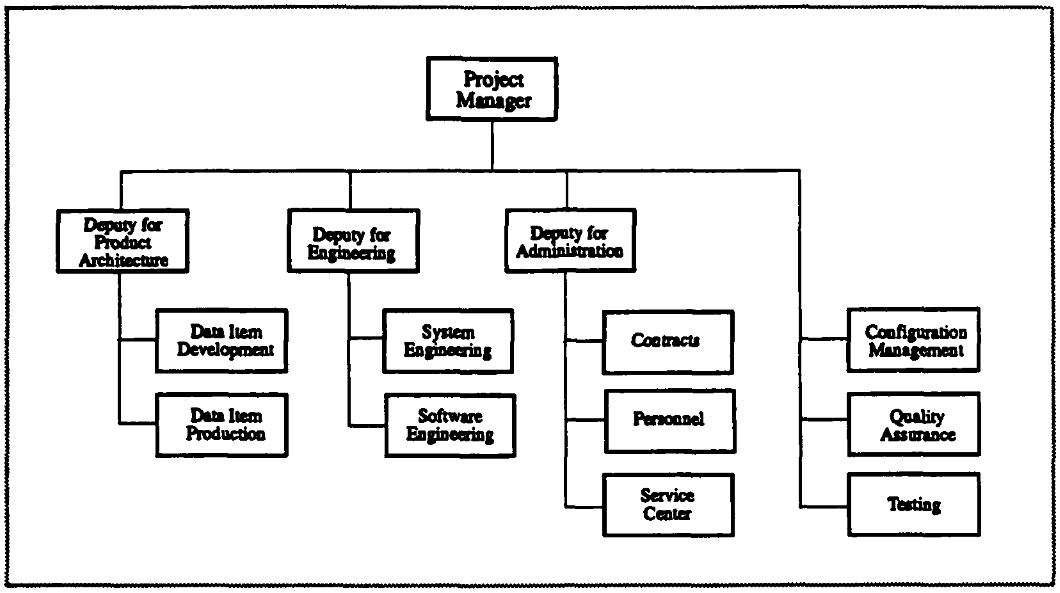


Figure 4. Sample Project Organization Chart Showing the Product Architecture Group

[4] DOD-STD-2167A, *Defense System Software Development*, U. S. Department of Defense, 29 February 1988.

[5] DOD-STD-2167, paragraph 3.12.

[6] DOD-STD-2167, paragraph 4.2.

[7] DOD-STD-2167, paragraph 3.7.

[8] DOD-STD-2167, paragraph 4.2.

[9] DI-MCCR-80028, "Data Base Design Document", paragraph 10.2.5.4, page 7.

[10] DOD-STD-2167, paragraph 5.3.1.2.

[11] DOD-STD-2167, paragraph 3.23.

[12] DI-MCCR-80028, "Data Base Design Document", paragraph 10.2.5.4, page 7.

[13] See DI-MCCR-80031, "Software Detailed Design Document", paragraph 10.2.5.3.1.3.1, page 10.

[14] The standard is vague on this point. The relevant passage appears in the DID for the SDDD, and it reads as follows:

> "In addition, if Unit Y is used in many places and resides in a library, this subparagraph shall identify: (1) the library by name and number and (2) the Software Detailed Design Document, by configuration name and number, in which the library description can be found (if not in this document)." (DI-MCCR-80031, "Software Detailed Design Document", paragraph 10.2.5.3.1.3.1, page 10).

[15] DOD-STD-2167, paragraph 4.2.

[16] DOD-STD-2167,paragraph 5.7.1.1.1.

[17] DOD-STD-2167, Appendix B, paragraph 20.4.5.2.c, page 70.

[18] DI-MCCR-80031, "Software Detailed Design Document", paragraph 3.1, pages 1-2. See also pages 5 and 7 in the same

[19] DOD-STD-2167, paragraph 5.7.1.1, page 39.

[20] DOD-STD-2167, paragraph 4.2.1.

[21] DOD-STD-2167, paragraph 5.3.1.2.

[22] McMenamin, Stephen M. and Palmer, John F. *Essential Systems Analysis*. New York, NY: Yourdon Press, 1984.

[23] DeMarco, Tom. *Concise Notes on Software Engineering*. New York, NY: Yourdon Press, 1979.

[24] This association process is also called "implementing the software organizational structure in Ada".

[25] See, for example, AFSCP 800-43, *Software Management Indicators*, Air Force Systems Command, 31 January 1986, page 10.

[26] Gray, Lewis, "Design vs. Coding: The Special Case of Development with a Compilable Ada Design Language According to DOD-STD-2167," unpublished paper, discussed at meetings of the ACM SIGAda Software Development Standards and Ada Working Group (SDSAWG), July 1986, November 1986.

[27] ANSI/MIL-STD-1815A-1983, *Reference Manual for the Ada Programming Language*, U. S. Department of Defense, 17 February 1983, page 10-1.

[28] The project used a method for transitioning from structured analysis to object-oriented design that produced a candidate list of classes, objects and operations directly from the DFDs in the SRS. The method is described in Gray, "Transitioning From Structured Analysis to Object-Oriented Design."

[29] MIL-STD-483A (USAF), *Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs*, U. S. Department of Defense, 4 June 1985.

[30] Barnard, H. Jack, et al., "A Recommended Practice for Describing Software Designs: IEEE Standards Project 1016," in *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pages 258 - 263.

[31] Gray, Lewis, "Considerations When Implementing the DOD-STD-2167 'Static Structure': Mapping CSCIs, TLCSCs, LLCSCs and Units to Logical and Physical Components of an Ada Program Library," in OOD *Readiness Task Report, Volume II - Rationale*, Army WWMCCS Information System (AWIS), Project Management Office, Fort Belvoir, VA, 22 May 1987.

[32] See Gray, Lewis, "Understanding CSCIs, CSCs and CSUs," in this report, page 1-4.

[33] See DOD-STD-2167, paragraph 4.2.1, page 15.

[34] MIL-STD-483A, paragraph 170.4.3, page 113, and paragraph 170.9, page 118.

[35] MIL-STD-483A, paragraph 170.4.2, page 113.

[36] MIL-STD-483A, paragraph 170.4.2, page 113.

[37] MIL-STD-483A, paragraph 170.4.3, page 113.

[38] For a discussion of with clauses, see the *Reference Manual for the Ada Programming Language*, paragraph 10.1.1, page 10-2.

[39] See Parnas, David Lorge and Clements, Paul C, "A Rational Design Process: How and Why to Fake It," in *IEEE Transactions on Software Engineering*, IEEE, New York, Volume SE-12, #2, February 1986, section VII, page 256.

[40] Ibid.

[41] MIL-STD-881A, *Work Breakdown Structures for Defense Materiel Items*, U. S. Department of Defense, 25 April 1975, paragraph 4.10, page 7.

[42] See Maibor, David, "Insights and Observations on DOD-STD-2167 & 2167A," in this report, page 1-15. See also, Gray, Lewis, "Understanding CSCIs, CSCs and CSUs," page 1-8.

[43] Joint Logistics Commanders, Joint Policy Coordinating Group on Computer Resources Management (JPCG-CRM), Subgroup on Computer Software Management (CSM), "Software Development Under DOD-STD-2167A: An Examination of Ten Key Issues," invited paper presented at Tri-Ada '89, U. S. Department of Defense, 25 October 1989, pages 9 - 10, in this report.

[44] Gray, Lewis, "Design vs. Coding: The Special Case of Development with a Compilable Ada Design Language According to DOD-STD-2167," unpublished paper, discussed at meetings of the ACM SIGAda Software Development Standards and Ada Working Group (SDSAWG), July 1986, November 1986.

[45] David Maibor Associates, Inc., *Seminar on DOD-STD-2167 / 2168*, Spring '90 series.

[46] Firesmith, Donald G., *Issues and Subissues Report of the ACM SIGAda Software Development Standards and Ada Working Group (SDSAWG)*, Association for Computing Machinery (ACM), Special Interest Group on Ada (SIGAda), 27 October 1986.

[47] Golubjatnikov, Ole, *Defense Software Master Plan: A Mechanism for United States Information Systems Leadership*, A progress report and recommendations to the Software Working Group, Defense Acquisition Board, Science and Technology Committee, coordinated by the JTC 1, Subcommittee 7, USA TAG, 16 November 1989, page 5-47.

[48] MIL-HDBK-287, *A Tailoring Guide for DOD-STD-2167A, Defense System Software Development*, U. S. Department of Defense, 11 August 1989.

[49] Meyer, Charles A., Lindholm, Sharon C., and Jensen, Jack L., "Experience in Preparing a DOD-STD-2167A Software Design Document for an Ada Project," in *Proceedings: Tri-Ada '89*, Association for Computing Machinery, New York, pages 118 - 124, also in this report.

[50] DOD-STD-7935A, *DoD Automated Information Systems (AIS) Documentation Standards*, U. S. Department of Defense, 31 October 1988.

[51] DOD-STD-1467 (AR), *Software Support Environment*,U. S. Department of Defense, 18 January 1985.

Lewis Gray is the President of Ada PROS, Inc. an Ada technology transfer small business founded in 1989 that specializes in reducing the cost of defense software programs through consulting and training on the sensible use of DoD software development standards. As a senior staff engineer at TRW Federal Systems Group, he created and implemented Ada software development standards for the Army WWMCCS Information System (AWIS) Phase 1, and led such software engineering activities as the preliminary design effort, and CSC integration and testing. As the program's Assistant Technology Director in Phase 2, he led its technology assessment and technology transfer activities. Previously, at GTE Government Systems Group, he was Ada development methodologist for a large information management system. Prior to that, at INTELLIMAC, Inc., he was Deputy Program Manager and a technical team leader of the Tactical Ada DBMS (TAD)/Army Field Artillery Tactical Data System (AFATDS) subcontract. He earned the B.A. degree from Stanford University, and the B.A. (mathematics) and M.A. and Ph.D. degrees (philosophy of science/technology assessment) from Indiana University. He is a former National Science Foundation Postdoctoral Fellow (technology assessment). Dr. Gray is a frequent speaker on software development standards and Ada, and Chair of the Association for Computing Machinery (ACM) Special Interest Group on Ada (SIGAda) Software Development Standards and Ada Working Group (SDSAWG).

Readers can write to the author at Ada PROS, Inc., 12224 Grassy Hill Court, Fairfax, Virginia 22033-2819 (Internet address: adapros@grebyn.com)

AUTOMATIC PROGRAMMING SYSTEMS FOR Ada: THEORY AND PRACTICE OF OBJECT ORIENTED METHODS OF PROGRAM SPECIFICATION FROM REQUIREMENTS

William Arden

Telos Corporation
55 North Gilbert Street
Shrewsbury, New Jersey 07702

Abstract

An analytical model for an Automatic Programming System (APS) for Ada is discussed. The model draws upon the software technologies of neural nets, fuzzy logic, and object oriented knowledge representation. Object oriented methods are shown for analysis of requirements and design representation so that a software program might be specified in Ada language from user-stated requirements.

Introduction

Automatic Programming is a topic in Artificial Intelligence that generally centers upon developing systems that are capable of programming themselves. Due to the object oriented nature of Ada, the APS is formulatable by object oriented methods. This paper states objected oriented methods of program specification into Ada language from user-defined, object oriented requirements. That is to say that once user-defined requirements are formulated into object oriented requirements, the object oriented requirements may be used to generate executable Ada code. This is an aspect of automatic programming in that once the user-defined requirements are "known" by the system, the system can then automatically generate executable code.

Background

The APS discussed in the introduction is not known by this author to exist as a complete system. However, the requisite software technology for each of APS software components does exist. The author has been involved in technology development and investigation in the following areas:

1) Object oriented analysis
2) Object oriented design representations
3) Neural nets
4) Fuzzy logic
5) Knowledge representation/expert systems
6) Ada language
7) Computer inference.

This paper does not present a detailed nor exhaustive treatment of each of these software technology areas, but rather an approach to their systematic usage in developing an APS for the Ada language.

System Overview

The system overview depicted in Figure 1 is a preliminary design which integrates technologies that have been explored.

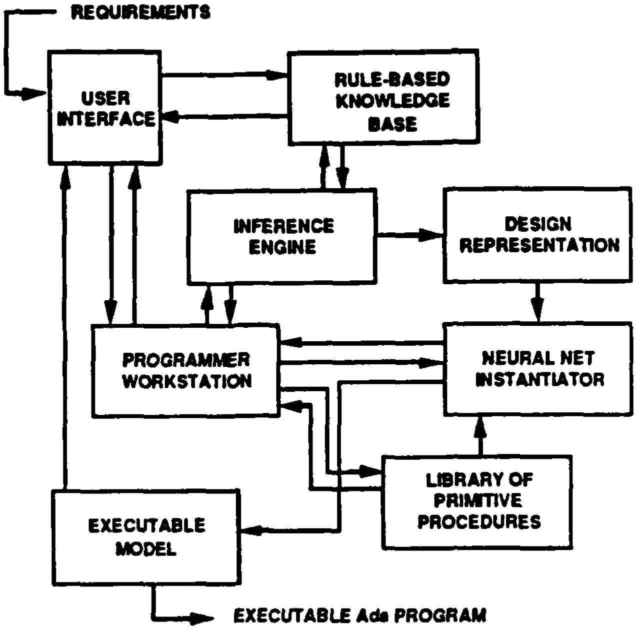


Figure 1. System Overview

A high-level description of the system is as follows:

1) The user interacts with an expert system comprised of a user interface, rule-based knowledge base, and inference engine to refine requirements into hierarchical object class relationships.
2) The inference engine then provides object control flow linkages between the objects of the knowledge base. This results in object control flow diagrams as a design representation.
3) The neural net instantiator then "fills out" the design representation from a library of Ada primitive procedures.
4) The filled-out design would be compiled, binded, and linked to produce an executable model that would be run at the user interface.

A more detailed description of the system, which explores the technologies of the components, follows in the succeeding sections.

System Components

User Interface

The user interface would be a rule-based windowing system. Windows would be hierarchical according to the levels of the requirements. The user interface would also allow windows for executable models, thus providing feedback on requirements specifications. An example rule depicting diagram is given in Figure 2. Figure 3 describes the hierarchical requirements windows.

Requirements Engineer: 1) Specify requirement A with restrictions of...
2) Specify requirement B with restrictions of...

APS User Interface: If requirements A and B are specified, what are the restrictions on requirements C and D?

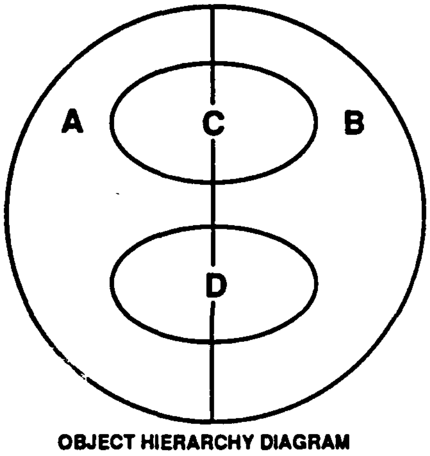


Figure 2. Rule-Based User Interface Iteration With Object Hierarchy Diagram Of Requirements

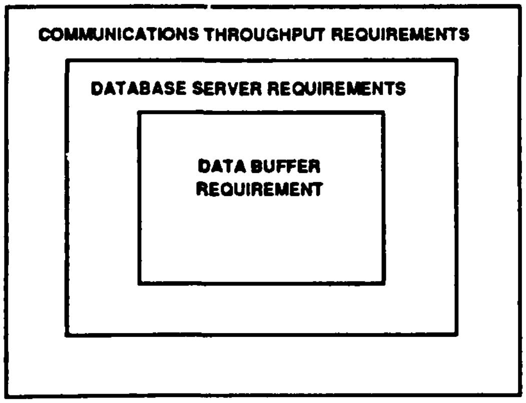

COMMUNICATIONS THROUGHPUT REQUIREMENTS
DATABASE SERVER REQUIREMENTS
DATA BUFFER REQUIREMENT

Figure 3. Hierarchical Requirements Windows

Knowledge Base

The knowledge base would consist of hierarchical object class frames. Each frame would consist of an object class, its subclasses, and the operations and attributes of the respective objects. Membership in classes would be by fuzzy set membership rules (with probability of membership in the class). The objects themselves are the result of an object oriented requirements analysis conducted by the Rule-Based System with the user.

Inference Engine

The inference engine would use fuzzy logic to refine knowledge base frames and also fuzzy logic to chain objects together into object control flow diagrams. This chaining procedure occurs when an object modified by its operation(s) has a fuzzy identity (@.90) with another object.

Design Representation

Design is represented by object control flow diagrams as shown in Figure 4.

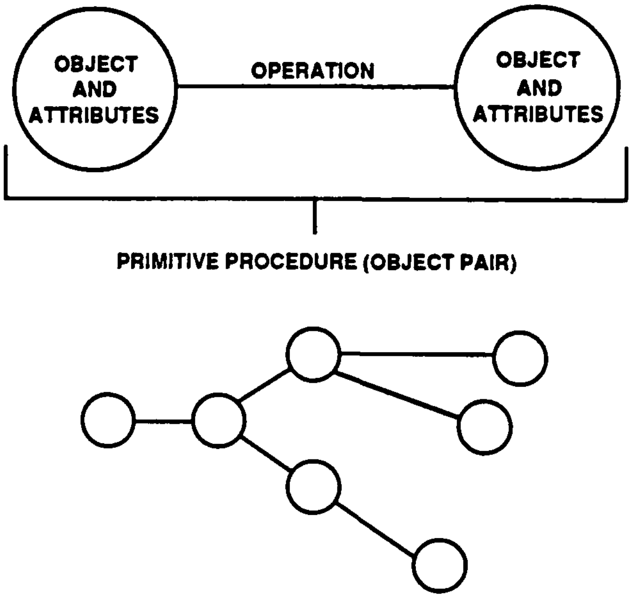


Figure 4. Object Control Flow Diagram (OCFD)

Neural Net Instantiator

The neural net instantiator would match object patch pairs (primitive procedures) with object pairs in the OCFD design representation. This neural net would generate a feature space of object attributes. The OCFD would be read for object attributes by a transducer program that updates the feature space for matched attributes. The neural net with updated features would complete missing features and then fire up from features to recognized object pairs. Once an object pair was recognized, its respective procedure would be mapped into the OCFD to form an executable model.

Executable Model

The executable model could be implemented as an Ada main procedure with all procedure calls and necessary declarations and a package of the primitive procedures recognized for the program.

Programmer Workstation

The programmer workstation would allow for an evolutionary development of the Ada APS by allowing the APS system programmer to make revisions and updates to the following:

1) User interface rules
2) Inference engine fuzzy logic
3) Neural net attribute/object pair connectionism generation
4) Library of procedural primitives.

Remarks

The system description given was purposely top level and component oriented. A detailed description of the component technologies is beyond the scope of this paper. The analytical model of the APS presented in this paper is intended to provide insight into methods of Automatic Programming along with Requirements Refinement and Engineering. Further, it suggests alternative methods of rapid prototyping and simulation of requirements.

General References

Object Oriented Programming Systems, Languages and Applications; Edited by Meyrowitz, Norman; ACM Press, 1989

Programmer's Reference Guide to Expert Systems; David Hu; Howard W. Sams and Company, 1987

Software Specification Techniques; Edited by N. Gehani and A.D. McGettrick; Addison-Wesley Publishing Co., 1986

Neurocomputing: Foundations of Research; Edited by James A. Anderson and Edward Rosenfield; MIT Press, 1988

About the Author

Mr. William Arden is a Principal Computer Engineer in the Advanced Software Technology Branch of Telos Corporation. His past work has included Objected Oriented Analysis, Specification and Design, Neural Nets, Expert Simulation Systems, and Requirements Engineering and Research. He holds a B.A. in Mathematics from Stockton State College, a MS in Electronic Engineering from Monmouth College, and is currently enrolled in a Ph.D. program in mathematics at Stevens Institute of Technology.

Address

William Arden
Telos Corporation
55 North Gilbert Street
Shrewsbury, New Jersey 07702

An Approach to Ada Implementation of an Associative Memory

E. K. Park and F. A. Skove

Computer Science Department
U.S. Naval Academy
Annapolis, Maryland 21402

C. S. Kang

Dept. of Computer Science and Info. Systems
The American University
Washington, D.C. 20016

Abstract

A simplified memory management scheme to solve problems with content-addressable memory (CAM) and associative memory (AM) is presented. CAM uses a parallel search. Thus, search time is drastically reduced. AM searches in the same way, except that relational functions are used. One problem with these techniques is cost of hardware. This paper proposes a software simulation of AM hardware. This will greatly reduce costs, as well as provide a ready prototype upon which to test application software for AM. Ada's Task construct was chosen for its powerful communication techniques between concurrent processes. Tasks immediately tell each other that the keyword has been matched to data in memory and searching ceases. This process cuts the search time by an average of one half compared to other concurrent constructs. Another problem addressed is that normally associated data cannot be accessed without first knowing the proper keyword for its parent element, resulting in the loss of access to some data. By allowing every memory cell to act as both an element and an attribute associated with another element, any information in memory can be accessed via a keyword match.

Keywords: Ada Task, associative memory, concurrent search, access and search time.

1 Introduction

Content-addressable and associative memory techniques are memory addressing methods that are unlike standard methods of memory addressing. Conventional memory storage is based on the storage of information in addresses. In a conventional memory storage system, to access any memory space the CPU must be able to determine an address for the desired data location.

Problems arise when using conventional memory storage architecture. Calculations involved in the determination of the effective address of a data word become time consuming. These addresses must pass serially through a bus and become a limiting factor when considering the speed of transmission through the bus compared with the speed of the CPU. The problem is known as the "von Neuman bottleneck"[1]. When parallel processing is involved, the effect is multiplied several times. Another disadvantage to conventional memory is found in a search-and-compare problem[2]. To search through a list and compare, an exact match will be found, on average, halfway through the list. Thus, the s^e^arch time increases linearly with the size of the list. To remedy these problems, new memory techniques were developed[1].

Content-addressable and associative memory methods do not use addresses to store information in memory. Instead, they rely on key words and words associated with the stored data. As an example of CAM, a data word may be divided into three fields[1]: tag, bits indicate the type of field (whether empty or used); label, which is the key word to be matched, and the actual data. In a search for label, the computer searches all memory locations of the correct type, as defined by the tag field, then looks for a match of the key word with labels of that type. This is not sequential search; rather it is performed in parallel. Thus, the search is not slowed by the amount of information in memory.

AM is similar, but instead of being limited to exact matches of key words with labels, related information may be accessed through the use of relational functions. Relations may be direct, such as "less than," "greater than," etc., or they may be indirect. Indirect involves associations of objects with values by using an *attribute* field. Thus, the object "car" may be associated with the value "GM" because one of its attributes is "manufacturer." This is a more powerful method than CAM, because an exact match is not necessary for an association. However, because of its generality, it cannot be as specific as CAM in information retrieval.

2 The Problem

There are several problems with CAM and AM techniques. According to Chisvin and Duckworth[1], some of these problems are listed:

1. functional and design complexity of the associative subsystem
2. relatively high cost for reasonable storage capacity
3. poor storage density compared to conventional memory
4. slow access time due to available methods of implemen tation
5. a lack of software to properly use the associative power of the new memory systems.

An additional problem that we have discovered with AM is that associated data may only be accessed through the key field. That is, if we do not know the proper key, we cannot access its associated data. This is analogous to the problem encountered when trying to look up a word in a dictionary, but not knowing how to spell it. The information is there, but you cannot find it.

Our research is twofold. First, we design a simplified memory management system to implement a variant of an AM using current technology. Using this system, we will be able to store and retrieve information from memory and solve the last problem above (loss of access to associated data). Our second goal, directly related to problem 2 above, is to decrease the cost associated with AM hardware by using a software simulation with concurrent processing. This could be readily transferred to a multiprocessor machine[3]. We chose to use Ada's Tasks, but any language supporting concurrent execution could be used (see section 4). Indirectly, this approach will also provide a possible solution to problem 5 above by making a working AM model for software development. Our solution for an improved AM management system is divided into several parts. In section 3.1 we discuss the specifics of the AM structure we have designed. Section 3.2 discusses the issue of memory partitioning and why this is integral to our solution. Section 3.3 addresses the topic of using Ada Tasks to search each memory partition concurrently for matches to a keyword.

3 The Solution

Because of the lack of systems upon which to base application software, software development for the AM systems has been slow at best. Using our software-based system, we simulate a hardware implementation of AM. This could conceivably afford application software developers a prototype upon which to test their programs and expedite the resolution of problem 5, as defined by Chisvin and Duckworth[1].

3.1 Memory Structure

The basic memory structure is a cell with three fields: tag, which is a boolean value indicating whether the cell is currently being used; data, which stores the actual data to be matched; and a pointer to a list of other cells associated with those data values. Figure 1 shows one cell whose data are associated with the data in two other cells. Figure 2 shows a simplified view of a memory with 15 cells and their associated data. It can be seen from Figure 2 that memory is composed of many interconnected linked lists.

Each cell can be accessed directly by using a key that exactly matches the data values in its data field. If the *search for a keyword matches* the data field of an element in memory, the search is successful and the data values are returned with a list of associated data (see section 3.3 for more information).

The second way that data values can be accessed is indirectly, via the association list that follows data values found as a match to a key. Thus, each cell may be either an element, or data values associated with another element. Traditionally, AM allows associated data to be accessed only when a keyword search is successful and the search returns the matching element and its associated data. By allowing every cell to function both as an element and as associated data, associated data may also be accessed directly.

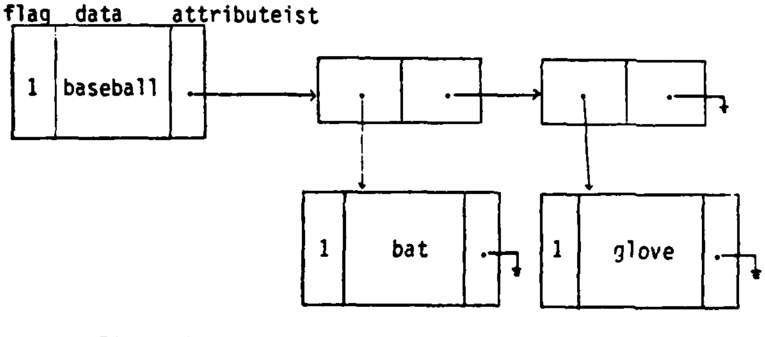


Figure 1. One cell with two associated cells

3.2 Memory Partitioning

To implement a parallel search requires either a multiprocessing system or a single-processor system with concurrent programming. Our design is based on a single-processor system with concurrent programming. Ideally, each process would search one memory location and the search would end in the time it takes to do one search. Unfortunately, this would require more processes than a typical system could support, given an average-sized memory. Memory must, therefore, be divided into equal size partitions, with one process per partition. Each process then searches its particular partition sequentially. Consequently, there are several parallel processes searching their own partition sequentially. The number of processes is determined by the system used.

Theoretically, for N processes, the total search time should be reduced by a factor of N. Because cpu throughput slows for an increasing number of processes, the theoretical search time is not attainable. Faster searches are realized when using a multi-processor system, but again, theoretical search speeds are not achieved. Section 3.3 addresses the issue of search times.

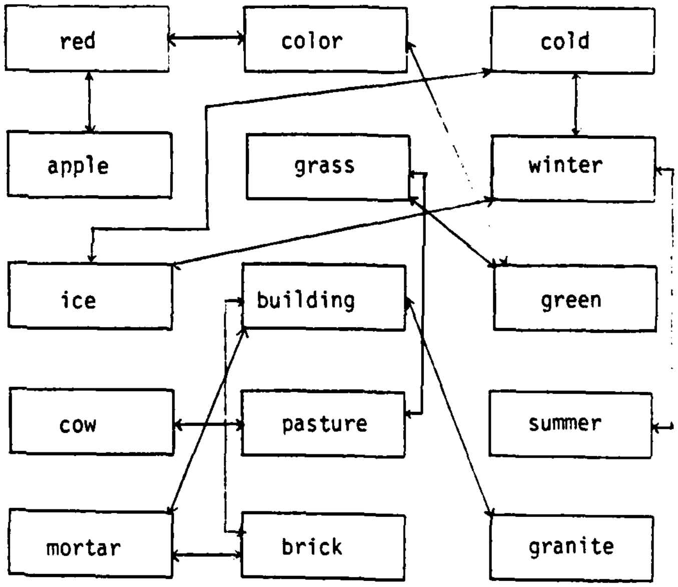


Figure 2. Sample memory with fifteen cells

3.3 Manipulating Memory

3.3.1 Searching Memory with ADA Tasks

We propose to use Ada's Task feature to concurrently search memory for a keyword. For example, to implement 10 parallel searches, we would partition the memory into 10 sections and use 10 concurrent tasks, each sequentially searching its own section of memory. This would speed the search drastically over a completely sequential search. However, requiring each concurrent process to completely search its memory partition is inefficient. Since, on average, a search is completed halfway through a list, search time could be reduced by one half if the search could be stopped when the keyword is found. Ada is ideally suited for this search because of the ability of its tasks to communicate with each other. Once a task matches a keyword to data in memory, a pointer to the data is saved and all other tasks are interrupted[4,5]. All data associated with the keyword are then returned. If no match is found, the tasks would return nothing and the search would fail.

A Task basically allows two or more sequences of actions to be performed concurrently. For example, the code

$$A := B(x) + C(y)$$

allows B(x) and C(y) to operate concurrently as opposed to sequentially. A can only be calculated by first calculating B(x) and C(y).

If a computer system has onl; one processor, the processor will allocate a specific time period to each task, very similar to a time sharing system. The processor alternates between the tasks operating by allowing each tasks to operate a few instructions, and then another, to make the system appear to be executing the tasks simultaneously. It is with this rationale that we decided to choose tasks to implement our memory management technique. As stated earlier, CAM and AM have slow access time due to the available methods of implementation. By using tasks to access our memory we will cut the time spent searching for memory cells considerably, since each task will be concurrently searching instead of sequentially waiting for each search to finish before starting.

Tasks communicate with each other using ENTRIES, ACCEPTS, and rendezvous. When executing a task, a simple ENTRY call is made to the task, which is very similar to a procedure call. An ACCEPT call "accepts" the entry call and ends the task's execution. For instance, assume the existence of a task called SEARCH. If Process1 makes an ENTRY call to SEARCH, then SEARCH is now executing. SEARCH must wait until an ACCEPT call is made by Process 1 to end its execution. Should Process 2 call an ENTRY to SEARCH while Process 1 is executing, it will be blocked. When Process 1 makes its ACCEPT call, the meeting point for SEARCH's ENTRY and ACCEPT is called the rendezvous. It is only when SEARCH receives its ACCEPT statement that the rendezvous is complete and SEARCH may be called again by another process.

In many situations, there is a need for several tasks to do the same thing simultaneously, such as perform the same search on different data in memory. Ada resolves this situation by providing task types. A task type means that all tasks of the same type have identical operations to complete. This brings us to our solution of the memory management system.

Initially, there is a given number N tasks (depending on the system implemented) and a memory block allocated for storage and retrieval of data. The memory must first be divided into separate partitions according to the number of tasks (N). Each task will have a specific partition to search. The task will be given exact memory locations according to memory addresses. The following code implements one possible solution to the concurrent searching routine that we have constructed:

```
TASK TYPE Search_Type IS
   ENTRY Search
      (Findname: IN Name;
       Foundname: OUT Pointer;
       Begin_Address: IN Address_Type;
       End_Address: IN Address_Type)
END Search_Type;
```

The type Search_Type allows us to make N tasks of the same type, since they will all be doing the same operations, but in different memory locations. Findname is the name we are searching for and Foundname is the pointer to the memory location of the Findname. Begin_Address gives the task the address of memory for which the task will begin searching, and End_Address is the address on which the task will end searching. Address_Type will be defined by the actual implementation used for memory locations.

When a task type is defined, it must be accompanied by a task body which describes the sequence of operations that the task will execute:

```
TASK BODY Search_Type IS
  Address: Address_Type;
BEGIN
  LOOP
    ACCEPT Search
       (Findname : IN Name;
        Foundname: OUT Pointer;
        Begin_Address: IN Address_Type;
        End_Address: IN Address_Type)  DO
- search memory between Begin_Address and End_Address
- until Findname is found
- when Findname is found tell all other tasks to stop
- searching
    END Search;
  END LOOP;
END Search_Type;

Task1, Task2, Task3,......,TaskN: Search_Type;
```

This declaration defines N independent tasks of type Search_Type. Each task has its own copy of the variables in the task body, but can communicate with the other tasks. Our main program looks like this:

```
BEGIN
  Task1.Search(Findname =>Cow;
        Foundname =>Ptr;
        Begin_Address =>0000;
        End_Address =>0100);
  Task2.Search(Findname =>Cow;
        Foundname =>Ptr;
        Begin_Address => 0101
        End_Address => 0200);
  .
  .
  TaskN.Search(....)
END;
```

3.3.2 Deleting information from Memory

There are three separate possible cases when deleting an element and/or some or all of its associated attributes. The simplest case is deletion of the link between an element and one of that element's attributes. The attribute is not deleted from memory, nor is the element. They are simply "disassociated." The algorithm for DELETE (ELEMENT (ATTRIBUTE-TO-BE-DELETED)) is:

1. Find ATTRIBUTE-TO-BE-DELETED in ELEMENT's attribute list
   a. Search each attribute of ATTRIBUTE-TO-BE-DELETED and check for link to ELEMENT
   b. Delete that link to ELEMENT and then reset link in the attribute list
2. Delete ELEMENT's link to ATTRIBUTE-TO-BE-DELETED

An example for DELETE (mortar (building)) is shown in Figure 3. The results of this operation on the sample 15-cell memory (Figure 2) are shown in Figure 4.

The next type of deletion removes an element from memory, and deletes all links between that element and its attributes. The attributes remain in memory. The algorithm for DELETE (ELEMENT) is:

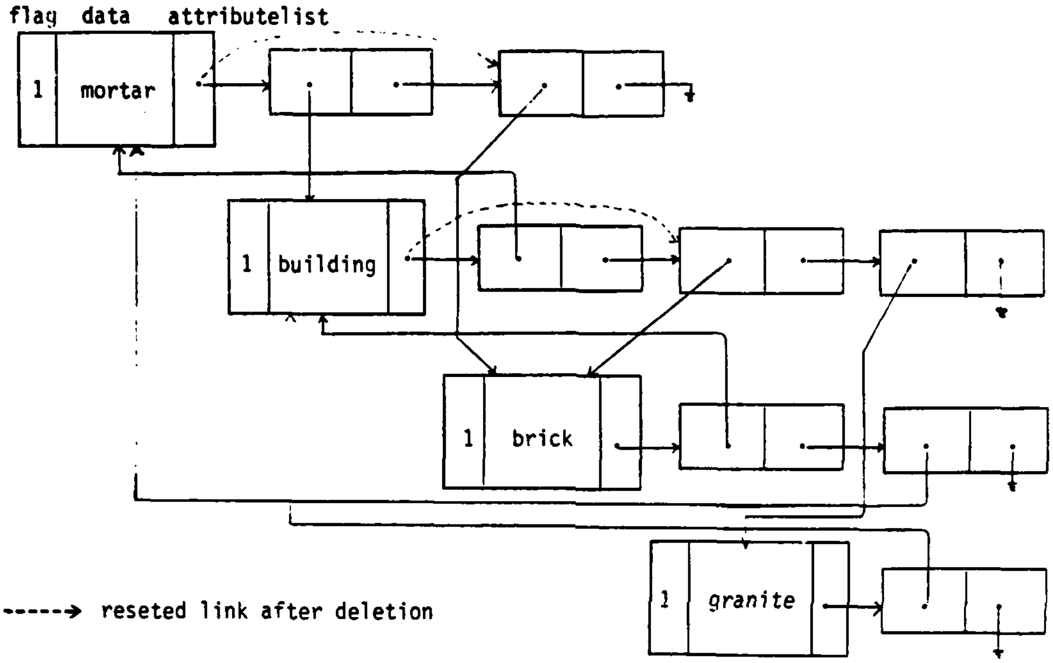


Figure 3. Deleting the attribute 'building' from the element 'mortar'

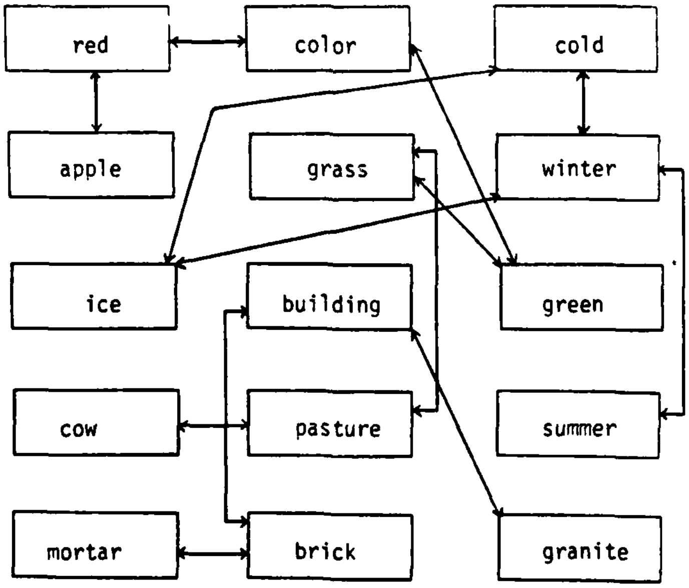
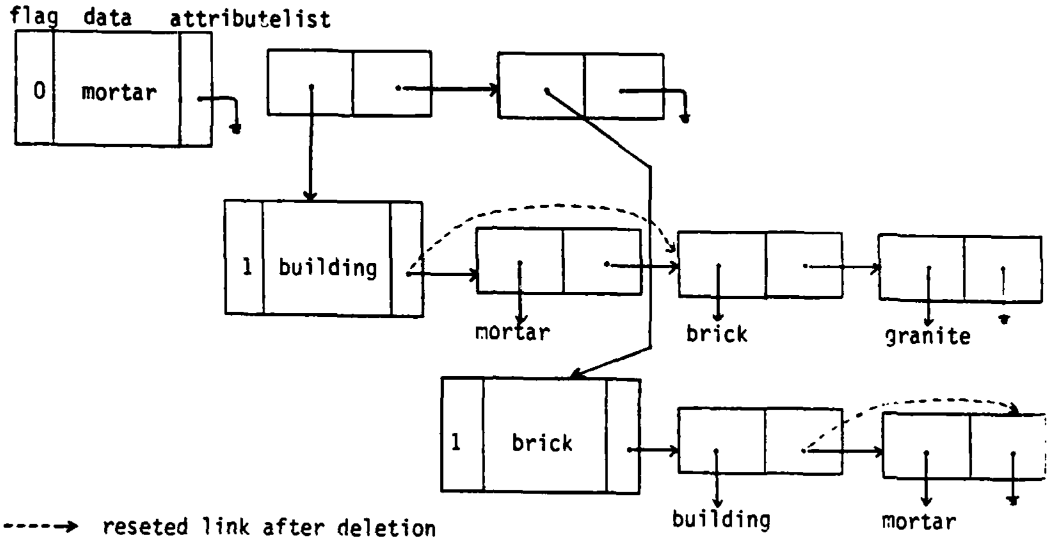


Figure 4. Sample memory afterDELETE(mortar(building))

1. For every attribute X of ELEMENT, delete every link to ELEMENT in the attribute list of attribute X and then reset link in the attribute list
2. Delete ELEMENT's attribute list (set **AttributeList** = null)
3. Delete ELEMENT (set Flag = 0)

An example for DELETE (mortar) is shown in Figure 5. The results of this operation on the sample 15-cell memory (Figure 2) are shown in Figure 6.

To implement a garbage-collection system, a list of free memory may be maintained. Free memory is linked by using the **AttributeList** field of each cell to point to the next free cell. Two additional pointers must be kept to point to the head and tail of the free memory list. Head-Free-Memory points to the first free memory cell at the head of the list. Tail-Free-Memory points to the last free memory cell, where newly freed memory cells are added. In the above example, step 2 of the above algorithm would be modified as:



Fiqure 5. Deleting the element 'mortar' from memory

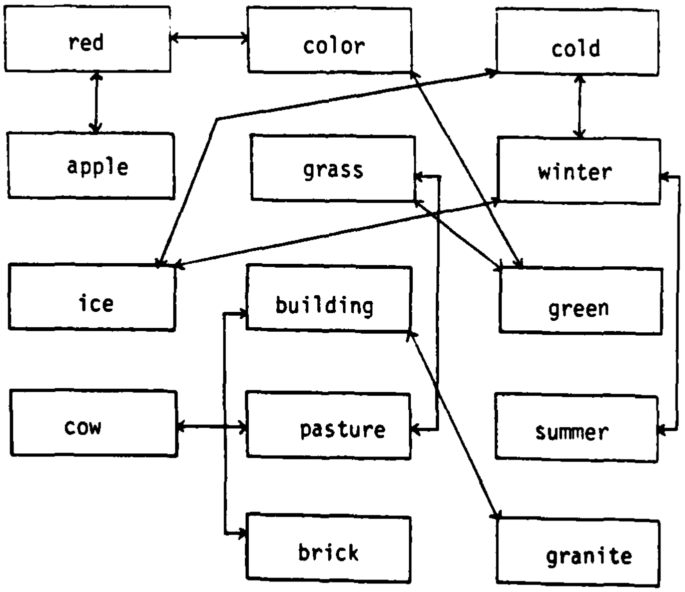


Figure 6. Samole memory after DELETE(mortar)

```
2. Delete ELEMENT's attribute list (set AttributeList
   = location pointed to by Tail-Free-Memory)
2a. Set Tail-Free-Memory = ELEMENT
```

Finally, a user may wish to delete an element and all of its associated attributes. This operation deletes the element, its attributes, all links between its attributes and each of their attributes. The algorithm for DELETE (ELEMENT) is:

```
1. For each attribute X of ELEMENT,
   a. Visit each of its attributes (except ELEMENT)
      and check attribute list and delete all links to
      attribute X or ELEMENT and then reset link in the
      attribute list
   b. Set flag of each attribute X to 0
2. Delete ELEMENT's attribute list (set AttributeList
   = null)
3. Delete ELEMENT (set flag = 0)
```

An example for DELETE (mortar) is shown in Figure 7. The results of this operation on the sample 15-cell memory (Figure 2) are shown in Figure 8. Of course, the same garbage-collection scheme can be implemented here.

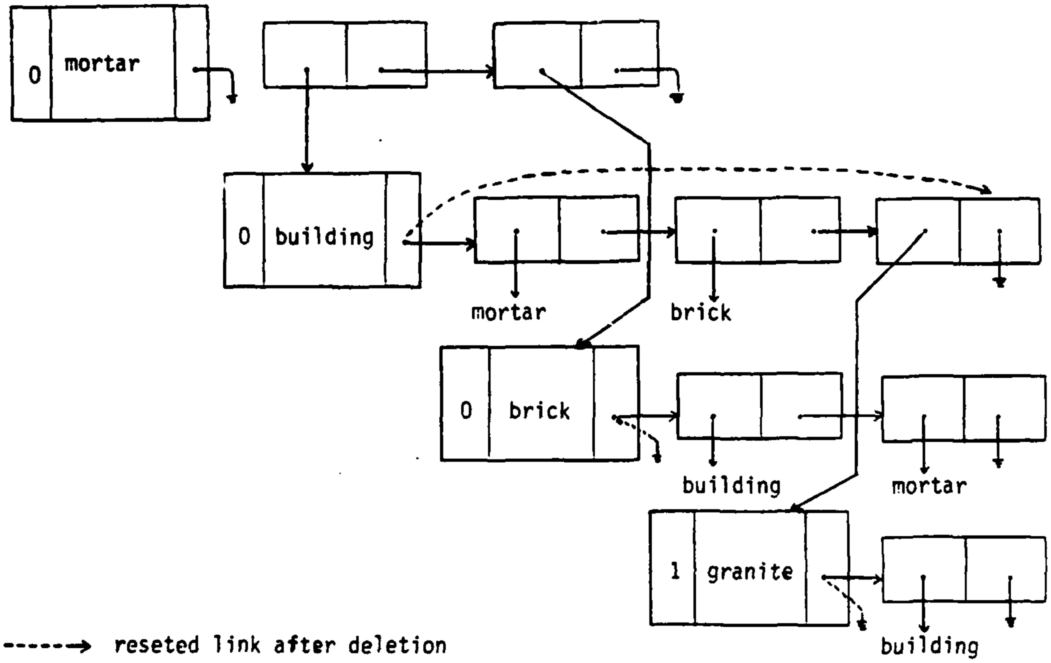


Figure 7. Deleting the element 'mortar' and all of its attributes

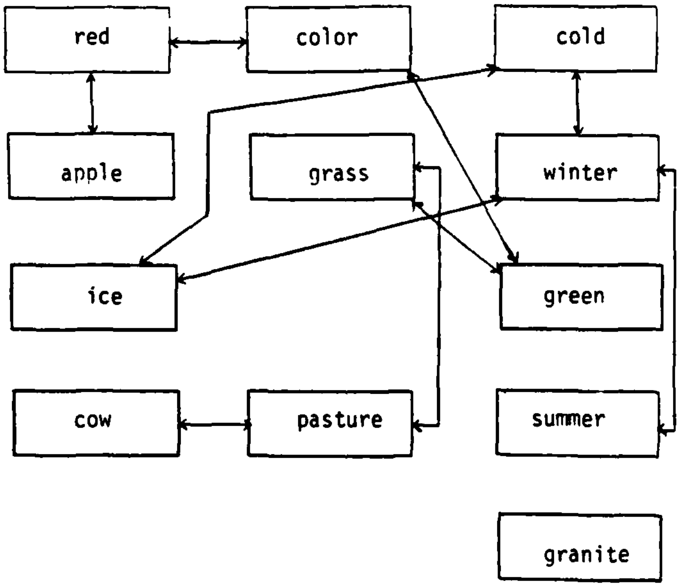


Figure 8. Sample memory after DELETE(mortar)

3.3.3 Storing Information in Memory

A single function suffices for each of the four cases encountered when storing information in memory. ADD (ELEMENT, ATTRIBUTE) satisfies the following cases:

(1) ELEMENT already exists and ATTRIBUTE does not,
(2) ELEMENT does not exist and ATTRIBUTE does,
(3) ELEMENT exists and ATTRIBUTE exists, but they are not associated with each other,
(4) ELEMENT does not exist and ATTRIBUTE does not exist

The algorithm for ADD (ELEMENT, ATTRIBUTE) is:

```
Search memory to see if ELEMENT exists (see Section 3.3.1)

1. If ELEMENT exists, search memory to see if
     ATTRIBUTE exists
   a. If ATTRIBUTE exists (case 3), establish links between
        ATTRIBUTE and ELEMENT
   b. If ATTRIBUTE does not exist (case 1),
      (1). Create ATTRIBUTE (from free memory list)
      (2). Add ATTRIBUTE to ELEMENT's attribute list
2. If ELEMENT does not exist, search memory to see if
     ATTRIBUTE exists
   a. If ATTRIBUTE exists (case 2),
      (1). Create ELEMENT
      (2). Establish links between ELEMENT and
             ATTRIBUTE
   b. If ATTRIBUTE does not exist (case 4),
      (1). Create ELEMENT
      (2). Create ATTRIBUTE
      (3). Establish links between ELEMENT and
             ATTRIBUTE
```

The diagrams associated with each case are omitted, but should be apparent from those provided for DELETE.

4 Comparison to Other Solutions

This section compares our solution to other possible implementations of associative search techniques. As noted earlier, Ada's Tasks are only one possible solution to our concurrency control problem. Tasks were chosen for this research for their ability to communicate between processes executing concurrently. With this approach, once a keyword is associated with data in memory, the task that found the data would immediately relay its discovery to the other concurrent tasks. Upon notification, all tasks would cease execution, allowing search time to be reduced by an average of one half.

Other possible concurrent programming constructs include Parbegin/Parend (specifically, Concurrent Pascal's Cobegin/Coend) and Fork/Join[6-8].

Parbegin/Parend is a simple construct that implements concurrent processes. Any statements between Parbegin and Parend are executed in parallel, while statements outside are executed sequentially. Concurrent Pascal's Cobegin/Coend is a specific implementation of Parbegin/Parend. A Cobegin statement simply marks N processes, each of which will concurrently execute to completion before the creating process is allowed to continue. Once the Coend statement is reached, execution of the main program is suspended and the initial procedures marked as concurrent are executed. These processes can execute in any given order, and often change from one run to another. Once all the concurrent processes have completed, the main program continues executing sequentially.

The restriction that concurrent processes cannot communicate with each other, as discussed above, limits the search speed because the concurrent processes cannot inform one another when one has successfully completed its search. Another restriction with Cobegin/Coend is that they cannot be nested. This limits application to simple cases. Fork/Join is a more powerful construct that can contend with all possible combinations of concurrent executions. It is not as simple as Parbegin/Parend due to use of gotos. The resulting "spaghetti code" can be both difficult to code and to debug.

5 Summary and Conclusions

This research for an improved AM system has advantages over traditional memory systems as well as conventional AM systems. Traditional memory search and retrieval time is relatively long due to the sequential nature of accessing physical addresses.

AM systems improve search and retrieval times by using a parallel search of a memory based on keywords. There are several problems with conventional AM techniques, however. One is the high cost of AM hardware. A possible remedy would be to use a software simulation. Ada tasks were chosen to simulate a parallel search by dividing the memory into N partitions, using one task per partition. When a task finds a match to the keyword for which it is searching, it communicates this to the other tasks and they stop their execution. This is an advantage that other concurrency constructs do not have; they must complete their processes before returning control to the process that initiated the search. The use of Ada tasks will, on average, be twice as fast as constructs such as Parend/Parbegin and Fork/Join.

Another common problem is the loss of access to information associated with a particular data value if the keyword is not known to the initiator of the search. This is solved by making each memory cell an element that can be accessed by a searching task. A cell can function as an element or as an associated data value. Associations are maintained through the use of linked lists.

This method would be ideally suited for a database system, and because of its logical separation of processes, could be easily adapted to a true multiprocessor system. It is far cheaper than its hardware counterpart, and could be used as a prototype for development of AM application software.

Acknowledgement
The authors would like to thank B. Hoyt and P. Moody for their time and report on the topic.

References

[1] Chisvin, Lawrence and Duckworth, R. James. "Content-Addressable and Associative Memory: Alternatives to the Ubiquitous RAM", IEEE Computer, Vol. 22, No. 7, July 1989, pp. 51-63.

[2] Bicault, Gary, "Juggling Multiple Resources", Byte, Vol. 13, No. 5, May 1988, p. 315.

[3] Park, E. K., Anderson, Paul, and Dardy, Henry, "An Ada Interface for Massively Parallel Systems", Proc. of IEEE International Computer Software and Applications Conf.(COMPSAC), October 1990, pp. 430-435.

[4] Habermann, A. Nico and Perry, Dewayne E., Ada for Experienced Programmers, Addison-Wesley Publishing Co., Reading, Mass., 1983.

[5] Cohen, Norman H., Ada as a Second Language, McGraw-Hill, Inc., New York, 1986.

[6] Deitel, H.M., Operating Systems, Addison-Wesley, Second Edition, Reading, Mass., 1990.

[7] Ben-Ari, M., Principles of Concurrent Programming, Prentice-Hall International, Englewood Cliffs, N.J., 1982.

[8] Graham, G. S., Holt, R. C., Lazowska, E. D., and Scott, M. A., Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley Publishing Co., Reading, Mass., 1978.

ADDING PERSISTENCE AND GARBAGE COLLECTION WITHIN ADA

Steven. J. Zeil

Computer Science Department
Old Dominion University
Norfolk, VA 23529

Abstract

The *Persistent Heap* abstraction eliminates the need for nearly all non-interactive I/O code. Objects allocated on this heap are accessed via *pointer* types analogous to Ada's access types. Any objects created or modified during the current program execution are automatically saved on secondary storage prior to program termination. Objects created by prior executions are automatically read into memory when a pointer to them is dereferenced. From the programmer's viewpoint, there are no I/O operations per se. A program simply traverses/accesses its abstract data types in the conventional manner without worrying about whether the data components are in memory. The Persistent Heap also provides for garbage collection of unreachable objects in both primary and secondary storage.

Introduction

This paper describes an experiment aimed at providing Ada programmers with a simple mechanism for programming with persistent objects in a managed storage space.

Persistent objects are identifiable objects whose useful lifetimes may extend beyond the execution of the programs that create them. Ada, like most languages, provides a standard file-oriented approach to persistence. Among the drawbacks to this approach are

- Different operations are required for manipulating persistent and transient (i.e., non-persistent) objects of the same type.
- When used in conjunction with strong file types, generic data structures are not easily provided with I/O operations.
- Programmers must often supply explicit code for translation and/or linearization of elaborate data structures.

There is a significant temptation in practice to design abstractions without provision for persistence, presumably with the intention of adding I/O later if circumstances require it. After-that-fact addition of I/O operations, however, may be quite difficult. Frequently it requires the introduction of a number of new data types ("external forms") capable of persistence and the introduction of code to map between the internal and external forms. In severe cases, the mapping may not be possible without alterations to the internal form (e.g., adding back pointers or other information to aid in linearization). Once completed, the additional data types and code required for I/O may pose a maintenance problem, since even small changes in the internal form must be reflected in the external form and may require significant changes in the mapping code between the two forms.

An interesting alternative to traditional programming-language I/O features is provided by "database programming languages", which attempt to merge the power of traditional programming languages with the storage handling capabilities conventionally associated with database systems[3]. The work described in this paper differs from these languages in that it emphasizes unobtrusiveness to the "traditional" programmer. Some essential features from other database programming languages have been preserved, such as simple forms of transaction management, but the type system and data manipulation primitives are drawn entirely from traditional programming languages. In particular, we have rejected the tendency of most database programming languages to organize persistent data into special "bulk" data types, such as sets or relations[6,9]. Instead, we advocate taking full advantage of the facilities for type construction and data abstraction already present in Ada. In this, our approach most nearly resembles that taken in PS-algol[1,2].

The work described in this paper was motivated by the author's work on language-processing and analysis tools for the Arcadia[11] and TEAM environments[4]. It was apparent from examination of the Ada code for these tools and of the error logs that issues of I/O and storage management were occupying an uncomfortably large portion of the implementors' time. I/O was a problem for the reasons outlined above. Storage management represented an additional, related problem. Primary-memory management of persistent objects is complicated by the fact that one cannot safely deallocate an object that may be persistent without first writing its current value to secondary storage. Furthermore, the most common basis for deciding when to explicitly deallocate an object (i.e., upon leaving the scope in which the object was declared) is by definition inappropriate for persistent objects. Finally, the application area (language processing and analysis tools) added further complications because of the preponderance of directed graph structures, many of them cyclic, for which simple schemes such as reference counting would prove ineffective. Secondary-storage management was also a concern, because the nature of persistent objects is such that they tend to be shared among a variety of tools, making it difficult and dangerous for any one tool to conclude than an object could be deleted from the secondary store. This difficulty is compounded by the possible later addition of new tools to the environment, utilizing the same objects, but unanticipated at the time of creation of the earlier tools.

Our goal was therefore to establish a simple means of adding persistence and storage management to our code, both the already developed Ada code and new code under development. To minimize intrusiveness, we sought a facility that could be used

This work was supported in part by grants NAG 1-439 from NASA Langley Research Center and CCR-8902918 from the National Science Foundation.

	Operation	Description
Transactions	Begin_Session	Start a transaction.
	End_Session	Commit a transaction.
	Abort_Session	Cancel a transaction.
Garbage Collection	Begin_Access End_Access	Protect local references.
Misc.	Select_Persistent_Store	Names the database in which to find the persistent heap.
	Termination	End-of-program clean-up.

Table 1: Persistent Heap Operations

	Operation	Description
Basic	Deref	Dereference a pointer.
	CDeref	Dereference a pointer, changing the object.
	Assign	Copy a pointer.
	=	Compare pointers.
Persistence	Name_Persistent_Object	Bind a mnemonic string to an object.
	Get_Named_Object	Retrieve the bound object.
	Retract_Name	Unbind a mnemonic.
Misc.	Self_Hash	Hash on ptr value.
	Protect	Protect an object.
	Touch	Communication protocol.

Table 2: Persistent Pointer Operations

preferably within the Ada language, or via a simple pre-processor from an Ada-like language into Ada, making only lexical and local syntactic alterations to the code.

Our approach centered upon the development of a *Persistent Heap* abstraction, which could be implemented using Ada's conventional abstraction and generic mechanisms. We were able to demonstrate an implementation of this abstraction for sequential code in which no pre-processor was required, though a simple one would have added some additional convenience. Supporting this same abstraction for concurrent code, however, turned out to be impractical without resorting to a more involved pre-processor, due to a surprisingly subtle feature (or lack of one) in Ada.

The next section of this paper introduces the Persistent Heap abstraction that forms the basis of our data model. After that, we describe the issues and design decisions governing the implementation of that abstraction, providing a sketch of our approach to adding both persistence and garbage collection without alteration of the underlying run-time system. The paper concludes with a description of our experiences using the sequential version to support the development of a number of programming environment components, and describes the difficulties we encountered in extending support to concurrent code.

The Persistent Heap

In seeking to provide persistence and garbage collection, a focus upon the "heap" is appropriate for many reasons:

- The conventional heap provided by Ada is not required to provide garbage collection[5], and commercial compilers do not, as a rule, provide it.
- Persistent objects must be identifiable in some manner. The conventional language mechanisms for identifying objects include variable names and pointers. The former are of use only for objects local to some scope activation, a natural view with which we had no wish to tamper or justification for doing so.
- Pointers (access types) are the only primitive language type or type constructor in Ada that is not subject to I/O.

The Abstraction.

There is agreement that a natural model of persistence within a language should offer persistence independent of type — objects of all types declarable within a language should be equally capable of persisting[3]. We therefore sought to provide an abstraction of "heap" and "pointer" that would alleviate the conventional prohibition against persistence of pointers and, by implication, of all types implemented using pointers. These abstractions are characterized by the operations showr in Tables 1 and 2.

The operations provided on the Persistent Heap (Table 1) are primarily concerned with establishing and managing the binding between the in-memory portion of the heap and the accumulated database of persistent objects on secondary storage. Hence, operations are provided for naming the database to be employed,[1] simple transaction control, and for protecting pointers local to a

[1] Currently, programs may only access a single database at a time, and inter-database references are not supported. An earlier version of the system actually did not impose these limitations, but the additional complications this introduced into the design of the tool built using this system prompted a retreat on this issue until a better interface for multi-database transactions and a multi-database name space could be agreed upon.

```
type Ptr is access T;
P1:  Ptr;
X, Y: T;
begin
   P1 := new T;
   X := P1.all;
   P1.all := Y;
```

a) Conventional Heap

```
package Aleph_T_Pointers is new
   Persistent_Pointers (T);
type Ptr is new Aleph_T_Pointers.Pointer;
P1:  Ptr;
X, Y: T;
begin
   Assign (P1, Allocate(new T));
   X := Deref(P1).all;
   CDeref(P1).all := Y;
```

b) Persistent Heap

Figure 1: Sample Pointer Manipulations

function/procedure (which will be discussed later in connection with garbage collection).

The operations provided on pointers into the persistent heap (Table 2) are intended to closely mimic those available on conventional pointer/access types. Thus, pointers can be dereferenced to yield the objects they identify,[2] can be copied, and can be compared for equality. Operations are also provided for binding mnemonic names to selected persistent objects in a fashion similar to that provided by PGraphite[12]. A hash function is provided on all pointers because our experience has shown that one of the major reasons for programmers devising "do-it-yourself" types logically equivalent to pointers has been the inability to use pointers within data structure implementations requiring faster-than-sequential searches. An operation is provided to protect selected objects from garbage collection, as will be discussed later. Finally, a Touch operation is used for all pointer types and types that contain pointers. This operation implements an inter-type communication protocol that is used by both the persistence and garbage collection mechanisms.

Ideally, a program employing the Persistent Heap should differ from a conventional Ada program primarily in the replacement of the built-in Ada access types by persistent pointer types. Figure 1a shows some typical manipulation of pointers on the conventional Ada heap. Figure 1b shows the equivalent manipulations of the Persistent Heap. Short as it is, this example demonstrates the most commonly used operations on persistent pointers. Like conventional access types, persistent pointers are strongly typed and can be dereferenced to yield only a single object type.

To permit the tracking of references needed for garbage collection. Persistent pointers are a limited private type. Thus they cannot be assigned using the standard Ada ":=", but the procedure **Assign** is provided as a substitute.

For portability reasons and for efficiency in dealing with accesses to small components of large objects, the Persistent Heap is built using Ada's primitive heap allocation and deallocation mechanisms. New objects are created by surrounding a conventional Ada "**new**" expression by a call to **Allocate**.

Pointers are dereferenced via the functions **Deref** and **CDeref**. The difference between these two operations is that **Deref** is used to access a component of an object, but **CDeref** is used to alter a component of an object. The two cases must be distinguished for persistence reasons, so that among the large numbers of persistent objects that may be employed during an execution, the usually small fraction of those that have been altered and must be written to secondary storage can be determined.

Type Hierarchies.

The benefits offered by the Persistent Heap tend to grow dramatically with the depth and complexity of the type hierarchy being manipulated by the program. The code in Figure 1, while it served to illustrate the basic operations offered by the Persistent Heap, is not representative. As a more typical example, Figure 2 illustrates portions of the program transformation system for the TEAM environment[4]. Figure 2a presents the declarations of some of the types employed by this system.

- Type **Transform_Sets** describes sets of **Transforms**, and is implemented using a reusable **Sets** generic. This generic employs the Persistent Heap internally in defining the type **Set**.
- Type **Transforms** provides a 3-tuple of **Patterns**, called the "match", "copy", and "re-examine" patterns, respectively. The Touch operation must be defined for any type on the Persistent Heap. For non-pointer types, **Touch** simply invokes the corresponding **Touch** procedure for each component type, passing the **Code** value along. (The **Codes** type is limited private with no operations visible to the programmer.) For pointer types, the **Touch** procedure is provided by the **Persistent_Pointers** generic.

  The "**Assign**" and other operations mentioned in the instantiation of the **Sets** generic are all derived from the **Pointer** type.
- Type **Patterns** is a limited private type that represents a directed graph structure and is implemented elsewhere using the Persistent Heap. The primary constructor operation for type **Patterns** happens to be named "**Pattern**".

The problem is, given a description of the relevant patterns, to generate a set of transforms and to make that set, the transforms, and the patterns persistent. Figure 2b shows the code for this problem as generated by the front-end to the transformation system. For each transform to be created, the three patterns are constructed using the operations appropriate to their abstraction and then assigned to the corresponding component of a transform T. When completed, T is added to a set S. The set S is forced to persist by binding a persistent mnemonic string to it.

[2] Actually, our dereferencing operators yield a conventional Ada access type so that accesses to components of large objects are not penalized by passing the entire object.

```
type Patterns is limited private;
type Transform_Bodies is
   record
      Match :  Patterns;
      Copy :  Patterns;
      Re_Examine:  Patterns;
   end record;

procedure Touch (
   Code:  in Persistent_Heap_Communication.Codes;
   Within:  in out Transform_Bodies);

package Transform_Manager is new Persistent_Pointers (
   Managed_Type => Transform_Bodies);

type Transforms is new Transform_Manager.Pointer;
package Transform_Sets is new Sets (
   Element => Transforms,
   Assign => Assign,
   Equal => ''='',
   Hash => Self_Hash,
   Touch => Touch);
```

a) Data Types

```
with Transformations; use Transformations;
procedure Set_Builder is
   T: Transforms;
      S: Transformations.Transform_Sets.Set;

      use Transform_Sets;

   begin
      Begin_Session;
      New_Set (S);

      Assign (CDeref(T).Match, Pattern (...));
      Assign (CDeref(T).Copy, Pattern (...));
      Assign (CDeref(T).Re_Examine, Pattern (...));
      Add (S, T);
        :
      Assign (CDeref(T).Match, Pattern (...));
      Assign (CDeref(T).Copy, Pattern (...));
      Assign (CDeref(T).Re_Examine, Pattern (...));
      Add (S, T);

      Name_Persistent ("name-for-the-set", S);
      End_Session;
      Termination;
   end Set_Builder;
```

b) Building a Persistent Set of Transforms

Figure 2: Persistence across Multiple Data Types

There is no user-written code to save the transforms within S or the directed graphs that comprise the patterns within those transforms. Nonetheless, before control has returned from the End_Session call, all of those objects will have derived persistence from S and will have been committed to secondary storage, simply because we have bound a name to S.

The Persistence Model.

The Persistent Heap hides all data storage within a transaction protocol[3] and all data access within the dereferencing operators.

At program termination, an object on this heap is automatically written to secondary storage if it is pointed to by another persistent object and if

- it was newly created during this program execution, or
- it's value was altered during the program execution.

Objects on secondary storage are automatically read into memory whenever a pointer to them is dereferenced by subsequent program executions.

Under these rules, persistence can be regarded as an inductive property passed from one object to another via inter-object references. Like any inductive property, special base cases are required from which other objects can derive persistence. In the Persistent Heap, these base cases are achieved by binding a user-supplied mnemonic string to an object. The bound object and all objects reachable from it derive persistence from the special, permanent, relation used to hold the (object,string) tuples. By querying that relation, the programmer can get access to the object serving as the root of a larger persistent structure.

These mnemonic bindings provide only a simple, flat name space for identifying objects, but are not expected to be employed very often. For example, a snapshot taken of the portion of the heap left on secondary storage after execution of several of the author's environment tools revealed some 50,000 objects of over 30 types, of which only 5 objects had been bound in this manner. Those 5 objects where themselves relations and tables that provided a hierarchical directory structure for a portion of the environment.

The Garbage Collection Model.

Automatic garbage collection has strong proponents in many conventional environments, but is especially important when dealing with large numbers of persistent objects. In part, this is because conventional rules by which a program might do its own collection (e.g., deallocate storage upon leaving the scope where the object was declared) are often by definition inapplicable to persistent objects. Also, by their nature persistent objects tend to be shared by a variety of programs. The possible addition of new programs interested in an object makes it difficult for any one program to safely conclude that the object may be destroyed.

Adding garbage collection on top of Ada, without reference

[3] Programs must invoke Begin_Session prior to any use of the persistent heap, and must later invoke End_Session to signal that changes to the heap are to be committed, or Abort_Session to indicate that the persistent heap should revert to its state prior to the start of the session.

to the underlying run-time system, is a challenge. Certainly, the possibility of employing limited private types to capture all pointer copying is a great help, but substantial problems remain:

- Garbage collection is, by its nature, an operation that spans many data types, making it difficult to implement in a strongly typed language.
- Although Ada permits type-specific initialization code, it does not support type-specific termination code. This raises problems in detecting the destruction of pointers locally allocated within some procedure activation that has since expired.
- Given the vagaries of expression evaluation orders, functions may return values, containing one or more pointers, that remain on the run-time stack for a very long time. Without access to the underlying run-time system, it is difficult to determine the useful lifetime of such references. The **return** statement is, in effect, a form of assignment that cannot be captured even by limited private types.

We therefore settled upon a scheme for assigning "ages" to objects on the Persistent Heap. Upon entry to a scope, we call a **Begin_Access** routine and just prior to exiting, we call **End_Access**. Each **Begin_Access**/**End_Access** pair defines a dynamic bracket of time. Objects created either locally or on the Persistent Heap are tagged with the innermost open bracket.

The set of nested open brackets at any moment during the program execution defines a discrete measure of the "age" of an object. Objects are immune from deallocation if they were created during a bracket that remains open or if they are referenced (directly or indirectly) by any object created during such an open bracket. This bracketing defines an alternate to the conventional root structure. Under this scheme, there is no "permanent" object from which references may be traced, but there is a set of objects of varying degrees of permanence.

A more obvious solution than the bracketing might have been to require an explicit termination operation on locally allocated objects upon leaving their defining scope. Such a termination operation could then signal the removal of any references held by the local objects. We prefer the bracketing scheme in part because it appears to lend itself more readily to an incremental approach to garbage collection, avoiding a potentially expensive overhead operation upon each procedure return. More importantly, however, the bracketing approach can more easily deal with references returned by functions.

If a function returns the value of a locally declared pointer, and that pointer had been the only reference to an object, we need to guarantee that the object will not be deallocated before that returned value gets used or copied into another pointer. There are two plausible approaches to that problem. Depending upon the exact garbage collection algorithm, we might be able to simply not use bracketing on functions, but let the local variables created by a function inherit their age from the most recently active procedure or block. This works reasonably well unless we have a deeply nested series of recursive function calls, in which case the amount of temporarily uncollectible garbage may become excessive. An alternative is to "protect" objects mentioned within function return expressions by moving them back one step in age, thus postponing possible deallocation of the referenced object by one bracket. This works well except in the case of functions that return large arrays of elements that in turn involve pointers, since every pointer must be visited in turn to alter the age of the referenced object. In programming directly with the Persistent Heap in Ada, we have employed a mixture of these two approaches.

Implementation

Persistence.

Each object for which an in-memory persistent pointer exists is represented by a "stub". The stub contains various management information and a conventional pointer (the *instance* pointer) to the actual object, which is created using the conventional Ada heap allocation mechanism. Although better performance could be obtained during allocation, garbage collection, and I/O by implementing our own memory management, such an approach would have encountered a number of problems:

- Allowing the dereferencing operators to return conventional pointers is a major asset in run-time performance. but might have been impossible if we used our own memory managerment.
- Portability would have been hard to maintain in the face of differing platforms requirements for boundary alignment. storage of discriminants. array dope vectors. etc.
- Providing for the initialization of allocated values would have been difficult or impossible.

The Persistent Heap is built on top of a standard interface[10] for object storage managers based upon a bytes/slots storage protocol. A number of storage systems can serve as implementations of this interface. Although we have implemented our own body for this manager, we hope later to employ more sophisticated object managers such as MNEME[8]. Under this interface, objects are stored as a block of bytes exactly reflecting their in-memory value, and a set of slots, one for each persistent pointer within the object. Each slot holds a secondary-storage inter-object reference.

Each persistent pointer has two components. The *self* component is a conventional pointer to the stub of the object containing that persistent pointer, and the *ref* component is a conventional pointer to the object being referenced. When a persistent pointer is dereferenced, the *ref* component is used to find the stub of the referenced object. If that stub's instance pointer is null, then the referenced object is not yet in memory and must be read from secondary storage. As each slot is read, the corresponding pointer is filled with the current location of the object referenced in that slot (this may require creating a new stub, if the referenced object is not yet in memory). Because objects are not read until a pointer to them is actually dereferenced, any amount of pointer copying, comparison, and other manipulation short of dereferencing can take place without actually bringing the object into memory.

Changed persistent objects are written to secondary storage as part of the **End_Session** processing or when the garbage collector determines that no in-memory references to them remain. An object's value is first written as a block of uninterpreted bytes. Then each persistent pointer in the object is visited, converted to a secondary-storage inter-object reference, and written into a slot. If the referenced object was created during this program execution and is not yet on secondary storage, then it is marked for subsequent output.

Garbage Collection.

A number of different implementation approaches are possible, given the bracketing model for garbage collection. Reference counting is an admissible approach, though it does not take full advantage of the bracketing. In a reference-counting scheme, when an object is deallocated the Touch protocol is used to send a "decrement-count" message to each object referenced by the disappearing one.

Conventional mark-and-sweep (preferably an incremental form) is also possible, if for each open bracket we keep a list of objects created during that bracket add to that list any objects referenced by a locally declared pointer during that bracket. To distinguish between pointers within objects on the heap and pointers within objects on the stack, we define the persistent pointer type's *self* component to have an initial value of some dummy object created during the innermost open bracket. This initial value is left unaltered for objects not on the Persistent Heap. For objects newly allocated on the Persistent Heap or read from secondary storage, the Touch protocol is employed to send a message to each pointer within the object to set its *self* component to that object's own stub (whose age is marked with the innermost open bracket).

A more intriguing possibility, however, is the use of *generational* garbage collection schemes[7]. Under these forms of garbage collection, objects are organized into successive *generations*. At periodic intervals, the most recent *k* generations are "condemned", where *k* is a random positive integer whose distribution tends toward lower values. Objects within condemned generations are retained (and possibly moved to an earlier generation) if they are directly or indirectly referenced by any object from an uncondemned generation. Instead of traversing the entire collection of objects to determine those that must remain, generational collection only "scavenges" through the condemned generations, usually a small fraction of the total store. Generational schemes are based upon the observation that objects end to become garbage very quickly, or else they tend to remain throughout the execution. Thus objects that survive a few rounds of garbage collection are not good candidates to become garbage in the future, and should not be examined as often.

Generational collection is a natural match to the bracketing of the Persistent Heap. Every time that a new bracket is opened, we can create one or more generations to hold the objects created during that bracket. During the time that a bracket is open, we simply prohibit the condemnation of the generations associated with that bracket. When the bracket closes, that prohibition is removed and the associated generations may be condemned.

Working with the Persistent Heap

Experience to Date.

The Persistent Heap is being employed in the development of language-processing tools for the Arcadia and Team environments[4,11,13]. Because these tools involve heavy use of a variety of directed graph structures, the garbage collection and automatic linearization capabilities have proven particularly appropriate.

A number of existing environment tools were ported to use the Persistent Heap, with the ported code amounting to about 20k non-comment Ada statements. The ported tools declared approximately 25 persistent pointer types. The ported source code was over 25 percent smaller than the original (by statement count), simply because of the elimination of explicit code for I/O and storage management. Furthermore, maintainance logs indicate that the eliminated code had been among the more trouble-prone portions of the system.

Perhaps more importantly, however, we find that the use of the Persistent Heap leads to simpler design for new objects. There is less temptation to complicate a data structure with extra identifiers and other "pseudo-pointers" whose primary purpose is to facilitate I/O or otherwise to ameliorate the weaknesses associated with conventional access types.

Against this must be weighed the code in terms of execution speed, executable size, and secondary store size.

- Certainly one cannot add the overhead of garbage collection without paying an execution time cost. Curiously, however, we find that our clock-time performance does not significantly degrade and, for some programs, actually improves when ported to the Persistent Heap. This appears to be a consequence of our own hardware environment's tending toward being I/O and network-bound, in which case a moderate amount of added CPU time goes unnoticed. Furthermore, because garbage collection reduces the amount of paging activity, a higher CPU utilization results in a lower clock-time.

  In other circumstances, e.g., a CPU-bound environment where quick real-time response is expected, our system would not fare as well, although its primary bottlenecks are inherited from the underlying secondary storage system (the generational garbage collection algorithm can be configured to guarantee a fixed maximal per-operation overhead[7]).

- Executable size is increased significantly because of our need to generate (via Ada generics) code for dereferencing, assignments, etc., for each pointer type.

- Secondary store size tends to be large, but this is actually more indicative of our particular implementation of the underlying storagemanager[10], which we intend to replace soon.

Concurrency.

Providing concurrent access to the Persistent Heap introduces further complications to garbage collection. The primary complication stems from the fact that simple bracketing provides a linear measure of object "age", but but concurrent object creation requires a number of simultaneous "time lines" for object ages.

If only a single process (Ada task) is creating objects, then the Persistent Heap need merely keep the current "time" in a variable and use that variable to initialize newly allocated objects on the heap and (pointers in) locally declared objects. This can be easily accomplished using Ada's built-in capabilities for object initialization, and can be accomplished without making the time variable visible or accessible to the programmer.

If more than one process is creating objects, then the time value used for those objects varies depending upon the process. A simple scheme would be to initialize each object using a function that returns the time appropriate to the process. Unfortunately, Ada does not provide any means for identifying which task is currently running (i.e., responsible for the function activation). The omission of any means of identifying tasks complicates matters tremendously. It means that a task identifier must actually be passed as an explicit parameter to every procedure and function. This parameter can be used to determine the proper initialization of local objects. This initialization, however, must occur within a critical region if Ada's built-in initialization mechanisms are to be employed, or must employ the Touch communication protocol at a much higher per-procedure-activation overhead.

The combination of the need to explicitly pass task identifiers to each routine and to embed local object initialization within a critical region was responsible for our conclusion that direct programming with the Persistent Heap in Ada was not practical for concurrent code. Instead, a pre-processor to generate the required Persistent Heap operations is required.

Such a pre-processor offers opportunities, however, to simplify both the process of creating persistent pointers (by hiding the generic instantiations behind a more conventional type declaration) and to simplify the manipulating code (by restoring the in-line syntax for assignment and dereferencing). The input language for this purpose is currently under development.

REFERENCES

[1] M. Atkinson, K. Chisholm, and P. Cockshott. PS-algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[2] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, Nov. 1983.

[3] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *Computing Surveys*, 19(2):105–190, June 1987.

[4] L. A. Clarke, D. J. Richardson, and S. J. Zeil. TEAM: a support environment for testing, evaluation, and analysis. In *Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments*, pages 153–162, Nov. 1988.

[5] Dept. of Defense, Ada Joint Program Office. *Reference Manual for the Ada Programming Language*. Government Printing Office. 1983. ANSI/MIL-STD-1815A.

[6] D. Heimbigner, L. Osterweil, and S. Sutton. *APPL/A: A Language for Managing Relations among Software Objects and Processes*. Technical Report CU-CS-375-87, University of Colorado. Department of Computer Science. 1987.

[7] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*. 26(6):419–429. June 1983.

[8] J. E. Moss and S. Sinofsky. *Managing Persistent Data with Mneme: Issues and Application of a Reliable, Shared Object Interface*. Technical Report COINS TR88-30, University of Massachusetts, Amherst, Massachusetts, Apr. 1988.

[9] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, 1981.

[10] P. Tarr and C. Lin. Ada interface to an underlying storage-manager. internal Arcadia document, Dec. 1988.

[11] R. Taylor, , F. Belz, L. Clarke, L. Osterweil, R. Selby, J. Wileden, A.L.Wolf, and M. Young. Foundations for the arcadia environment architecture. In *Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments*, pages 1–13, Nov. 1988.

[12] J. C. Wileden, A. L. Wolf, C. D. Fisher, and P. L. Tarr. PGRAPHITE: an experiment in persistent typed object management. In *Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments*, pages 130–142, Nov. 1988.

[13] S. J. Zeil and E. C. Epp. Interpretation in a tool-fragment environment. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 241–248, IEEE, Apr. 1988.

Steven Zeil received his Ph.D. in 1981 from the Ohio State University Department of Computer and Information Science. He is currently an Associate Professor in the Computer Science Department at Old Dominion University. His research interests are software testing and software development environments. Inquiries regarding this paper may be sent to him at there or by internet to zeil@cs.odu.edu.

ANALYSIS OF SOFTWARE REUSE
ON AFATDS CONCEPT EVALUATION

by C. Alan Bumham, Robert D. Gerardi, Peter Ho, and Dr. Harry F. Joiner II

Telos Corporation
55 North Gilbert Street, Shrewsbury, NJ 07702

Introduction

The reuse of software, particularly on large Ada projects, has received significant attention recently. This study analyzes the Advanced Field Artillery Tactical Data System (AFATDS) Concept Evaluation Phase (CEP) software for reuse of code and discusses how the design of the system implemented a reuse strategy.

The AFATDS CEP software is composed of eight independent Application Program Units (APUs), or separate applications:

- Movement Control Tactical Operations (MC_TAC_P)
- Movement Control Fire Support Control (MC_FSC_P)
- Status Functions (SF_PROC)
- Fire Support Planning (FSP_PROC)
- Fire Support Execution, Fire Support Control (FSX_FSC_P)
- Fire Support Execution, Tactical Operations (FSX_TAC_P)
- Fire Support Execution, Target Processing (FSX_TGP_P)
- Data Base Display Function (DBDF_PROC)

Each of the APUs is a stand-alone system when loaded into the hardware, and no separate APUs can be loaded simultaneously on the same hardware unit. In essence, AFATDS is not a system, but eight systems that were developed together to support the field artillery command and control functions. This joint development represents the first aspect of the reuse strategy.

For the purpose of this paper, software reuse will be measured by the lines of code that the APUs have in common at the module level. Although no less important for the effort, the reuse of design and other software elements were not measured independently.

The software for the AFATDS CEP was developed by Magnavox Electronic Systems Company. This study is a part of an extensive source code analysis [1, 2, 3] commissioned by the Office of the Project Manager, Field Artillery Tactical Data Systems (PM, FATDS) and performed through the U.S. Army Communications-Electronics Command (CECOM), Center for Software Engineering (CSE) under U.S. Army Contract Number DAAB07-87-C-B015.

Design Approach

The development strategy included a number of elements to increase software reuse. A primary factor from the beginning was a conscious effort by the system design team to establish an architecture that fostered reuse and identify potential internal reuse candidates. This software execution environment organizes the common execution elements into 16 Process Libraries (PLs) and 26 User Libraries (ULs) for development. Some of the design techniques were standard, such as isolating the support functions associated with the operating system, communications support, and data management. Other common functions were identified as application-type functions and grouped together. This design effort clearly distinguished between the role of a software component during the development effort and the role that component plays during execution.

The AFATDS Project Library General Compilation Guide (APLGCG) is illustrated in Figure 1. The APLGCG provides a network diagram that shows the order of compilation for the ULs, PLs, and APU command files. Each of the UL command files is represented by a rectangular block in the APLGCG. The PLs and APU components are shown as rounded blocks. The compilation order also indicated the dependencies at a library level.

At each stage of the design and development work, the team leaders met weekly to discuss details of their teams' current work for the purpose of identifying common software. When a part was identified as a requirement by more than one team, the software was developed to satisfy the needs of all of the teams that were expected to use that capability. The emphasis at this point was on minimizing the development of unique code. Some additional effort was normally required to make the software usable by more than one APU; however, this effort was clearly compensated for by the amount of reuse achieved.

Initially, the use of Ada generics was hampered by compiler performance. This restricted the use of a major Ada reuse feature. The developer compensated for this compiler problem by limiting the use of generics during the early development. The later development included extensive use of generics.

Managing for Reuse

The extensive reuse required careful planning by the developer for configuration management and compilation.

AFATDS PROJECT LIBRARY GENERAL COMPILATION GUIDE

LIBRARY VERSION: 4.32
ENVIRONMENT: TELESOFT Ada
STATUS: COMPLETE

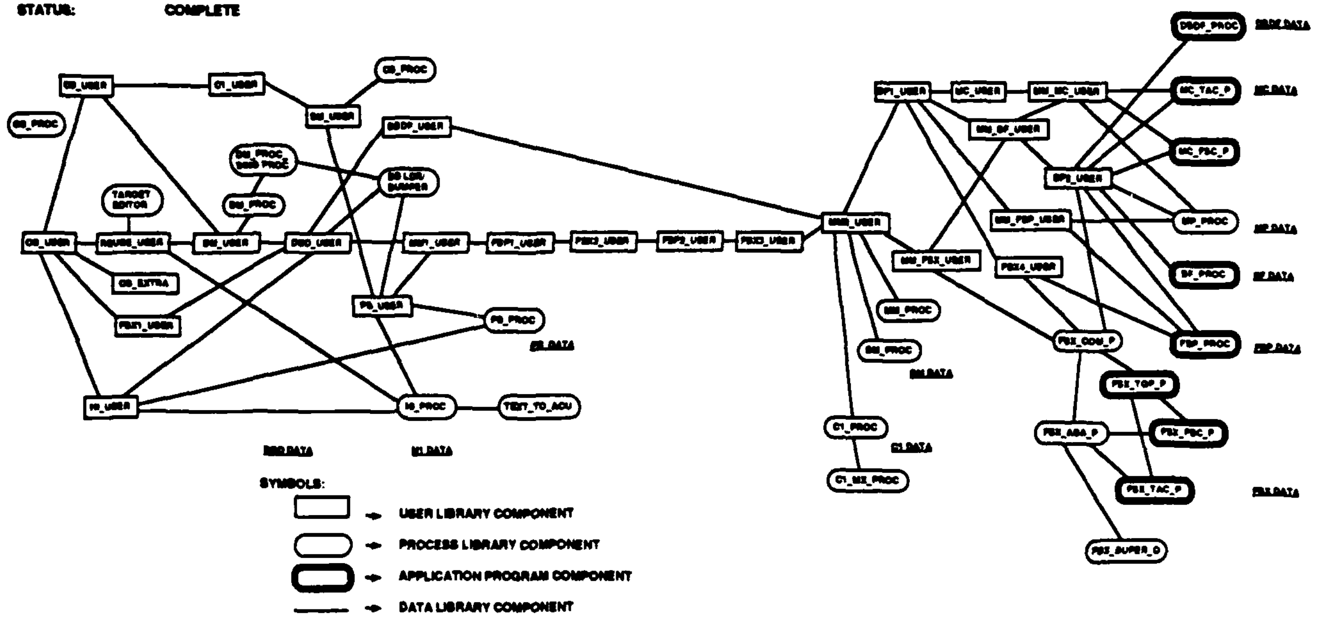


FIGURE 1

31 MARCH 1989
NOTE: NETWORK FLOW IS LEFT TO RIGHT

The project maintained thorough configuration control of reuse components, even though the corporate reuse group supported the reuse library. Multiple versions of the same reuse components could be used by the project if the configuration control is outside the project level.

The development management must establish a meaningful and realistic reuse plan as part of the software development plan. Strong management leadership and commitment were necessary to overcome the resistance of the developers. The managers insisted on reuse and inspected for it. Other key factors were management's acceptance of the responsibility when problems arose due to reuse and their willingness to improve the reuse software. The descriptions used for selection of reuse components had to be precise and accurate.

Confidence in reuse components is difficult to establish, but is easy to destroy. Designers and programmers resisted the reuse of software, suggesting that they could develop the needed capabilities faster and with greater efficiency themselves. The individual developers would not accept shortcomings in the reuse components that they would overlook in their own software components, underscoring the need for management intervention. They easily recognized the value of the reuse of complete subsystems and large scale fragments, but had more trouble accepting the need for small component reuse.

Analysis of Source Code for Reuse

The analysis was performed on Version E7 of the CEP software as delivered to the Government by the developer. The tape contained both the source code files and the VAX command files that are used to build the APUs. The more than 8,000 files were combined into two master directories. Lists of the source code files associated with each block of the APLGCG were created from the appropriate command files. The PL and UL blocks that are reused in all of the APUs are collectively referred to as the "APUCOM."

The source code was measured using SIZER, a tool developed by TELOS to measure the number of lines of code (LOC) in one or more files of Ada code. Three different LOC measures were made: terminal semicolons (TSC); noncomment, nonblank lines (NCNB); and carriage returns (CR). SIZER was executed to count the LOC for each of the UL, PL, and APU component files in the APLGCG. The results for APUCOM and each of the complete PLs were tabulated.

Tables 1, 2, and 3 provide the LOC measures for each of the component ULs, PLs, and APUs, respectively. The cumulative totals for the PLs, APUCOM, and the APUs are provided in Table 4. In Table 5, the percent of reuse in each of the APUs is presented in two ways: the percent of the APU that consists of APUCOM and the percent of the APU that is used by at least one other APU. TSC was used to determine these percentages. One of the APUs (FSX_TGP_P) is, in fact, a subsystem of FSX_FSC_P and FSX_TAC_P and thus consists entirely of reused code. Approximately 43 percent of the total amount of code used in the eight APUs was developed because of the internal reuse efforts on AFATDS.

Table 1. SIZER Lines of Code (LOC) Counts for the User Libraries

	SIZER LOC Count		
	TSC	NCNB	C R
USER LIBRARIES			
CI_USER	1421	4111	9472
CS_USER	16904	37436	81059
DBDF_USER	401	1243	4048
DBD_USER	6121	9045	10743
DM_USER	9639	29622	82586
FSP1_USER	1899	3606	9821
FSP2_USER	792	4979	6430
FSX1_USER	925	2243	4538
FSX2_USER	1170	2840	6374
FSX3_USER	2919	9965	22047
FSX4_USER	1921	5717	13730
HI_USER	3073	9162	20671
MC_USER	502	1086	2902
MC_USER32	698	1449	3892
MM1_USER	571	1593	4596
MM2_USER	8877	21575	48128
MM_FSP_USER	7617	23911	46638
MM_FSX_USER	10129	31096	61147
MM_MC_USER	1169	2873	4884
OS_USER	7316	14435	34640
PS_USER	954	2235	5670
REUSE_USER	3923	8122	24534
SF1_USER	2281	6401	14279
SF2_USER	3150	8306	13898
SM_USER	420	935	3041
TOTAL	94792	243986	539768

Table 2. SIZER LOC Counts for the Process Libraries

	SIZER LOC Count		
	TSC	NCNB	C R
PROCESS LIBRARIES			
CI_MX_PROC	9652	24468	44954
CI_PROC	16886	41649	80518
CS_PROC	14229	34552	68736
DB_LOADER_DUMPER	61820	152269	205944
DM_PROC	19926	61193	159110
DM_PROC_SING_PROC	1444	3366	7826
FSX_ASA_PROC	11966	38612	72769
FSX_COM_PROC	37351	118162	241056
FSX_SUPER_Q	228	592	1320
HI_PROC	29627	84996	186323
MM_PROC	971	1531	2231
MP_PROC	8365	23719	50184
OS_PROC	26527	56195	129916
OS_EXTRA	1051	2066	2560
PS_PROC	969	2153	4105
SM_PROC	19492	45831	112947
TARGET_EDITOR	1352	3210	5419
TEXT_TO_ACU	1835	6809	15487
TOTAL	263691	701373	1391405

Table 3. SIZER LOC Counts for the APU Components

		SIZER LOC Count	
	TSC	NCNB	C R
APU COMPONENTS			
DBDF_PROC	6617	19465	50756
FSP_PROC	99595	280901	561781
FSX_FSC_PROC	146	613	1016
FSX_TAC_PROC	135	516	875
FSX_TGP_PROC	6389	19539	38009
MC_FSC_PROC	8876	25226	62393
MC_TAC_PROC	15720	44663	108892
SF_PROC	19266	58720	109013
TOTAL	156744	449643	932735

Table 4. Cumulative LOC for the Process Libraries and the APUs

		SIZER LOC Count	
	TSC	NCNB	C R
PROCESS LIBRARIES			
CI_MX_PROC	91244	233793	500266
CI_PROC	81592	209325	455312
CS_PROC	57194	128905	278007
DB_LOADER_DUMPER	159578	419724	574299
DM_PROC	30804	113372	300870
DM_PROC_SING_PROC	49152	154174	390755
FSX_ASA_PROC	147492	411078	875369
FSX_COM_PROC	135526	372466	802600
FSX_SUPER_Q	113730	411650	876689
HI_PROC	43939	116715	266168
MM_PROC	65677	168207	377025
MP_PROC	105040	275371	599346
OS_PROC	26527	56195	129916
OS_EXTRA	8367	16491	37200
PS_PROC	44888	98741	219046
SM_PROC	84198	213507	487741
TARGET_EDITOR	12591	25773	64593
TEXT_TO_ACU	45342	122471	280271
APU COMPONENTS			
DBDF_PROC	102871	268052	598570
FSP_PROC	308043	662976	1829663
FSX_FSC_PROC	154027	331230	914394
FSX_TAC_PROC	154016	331746	915269
FSX_TGP_PROC	141915	392005	840609
MC_FSC_PROC	107499	279221	621885
MC_TAC_PROC	114343	298658	668384
SF_PROC	115520	307307	656827
TOTAL	1198234	2871195	7045601
APUCOM TOTAL	96254	248587	547814
TOTAL ALL CODE	515227	1395002	2863908
SUM OF APUs	1198234	2871195	7045601
CODE DEVELOPED (%)	43%	49%	41%

Table 5. Percentage of Reuse Based on APUCOM and All Reused Libraries.

	APUCOM Percent	Total Reuse
APUs		
DBDF_PROC	94%	94%
FSP_PROC	31%	68%
FSX_FSC_PROC	62%	100%
FSX_TAC_PROC	62%	100%
FSX_TGP_PROC	68%	95%
MC_FSC_PROC	90%	92%
MC_TAC_PROC	84%	86%
SF_PROC	83%	83%

Additional Reuse Efforts

Besides the reuse of software within the AFATDS program, Magnavox has reused substantial parts of the AFATDS CEP design and code on two other projects [4]: the Elevated Target Acquisition System (ETAS) Central Processor and the Navy Force Fusion System (FFS) Command and Control prototype. On ETAS, Magnavox reused code (over 150,000 out of 165,000 LOC), architecture and design, documentation, standards, and development environment. The FFS project was supported by the STARS program as a reuse demonstration and achieved over 93 percent reuse on more than 200,000 LOC [5]. The productivity on these projects proved the value of large scale reuse even more clearly.

References

[1] H. F. Joiner. "Metrics Analysis of Ada Programming Practices on AFATDS - a Large Ada Project," 43rd AFCEA International Convention. June 1989.

[2] C. A. Burnham, P. N. Ho, and H. F. Joiner. "Techniques for the Analysis of Portability - a Study of the AFATDS Concept Evaluation Code," Eighth Annual National Conference on Ada Technology. March 1990.

[3] P. N. Ho. Reuse Study on AFATDS Application Program Units. TCFM 89-5908 13 November 1989.

[4] R. Lawson. "Ada Reuse - Results and Issues," Development of Large Software Systems in Ada. 12 July 1990.

[5] H. F. Joiner. "Economic Analysis of Software Reuse," Eighth Annual National Conference on Ada Technology. March 1990.

Biographies

C. Alan Burnham is a Senior Systems Engineer with TELOS at Fort Monmouth, NJ. He is currently the Technical Task Leader for Ada Engineering Support.

Mr. Burnham has worked over 20 years in the development and support of software-intensive military systems. He has performed software development, software quality evaluation, systems analysis/engineering, software acquisition management, and program management. Mr. Burnham's recent experience has been primarily in developing specifications, defining requirements, and testing software for mission-critical defense systems. His earlier experience was in the development, validation, and application of large-scale, high-resolution combat simulations.

Mr. Burnham received his BA in Mathematics from Augustana College.

Robert D. Gerardi is a Systems Engineer with TELOS at Fort Monmouth, NJ. He is currently the support engineer for Ada Engineering Support. His main areas of interest are PC to PC and PC to mainframe communication links, Ada software metric and source code analysis, and PC system support.

Mr. Gerardi received his BA in Computer Science from Central Connecticut State University and is currently pursuing his MA in Computer Science from Monmouth College. Mr. Gerardi has over 5 years experience in the development and support of PC systems and communication links.

Peter N. Ho is a Senior Software Engineer in the Advanced Software Technology group of Telos Corporation. His main areas of interest are CASE tools, Software Reverse Engineering Technology, Ada Software Reuse Technology, and large-scale software metric analysis.

Mr. Ho received his BA in Mathematics from Lincoln University, MS in Mathematics from Lehigh University, and MSE in Computer and Information Science from University of Pennsylvania. Mr. Ho has over 10 years of experience in the development of software for both Government and commercial applications.

Harry F. Joiner joined TELOS as a Software Engineer in August 1986, working on the Firefinder field artillery location radar. He assumed his current position as Manager of Software Metrics in March 1988. TELOS is the largest software engineering support contractor to CECOM CSE.

Before joining TELOS, Dr. Joiner served as a consultant on project management and digital signal processing to various oil companies. He has over 20 years of experience in mathematical modeling, digital signal processing, engineering, and project management.

Dr. Joiner has a BA in Mathematics and Chemistry and an MS and Ph.D. in Mathematics. He has served on the faculties of the University of Massachusetts and Texas Christian University.

A METHODOLOGY FOR THE EVALUATION OF REUSABLE ADA SOFTWARE LIBRARIES

Stephen H. Levy
Calculemics, Inc.
100 Willoughby Rd.
Fanwood, NJ 07023
(201) 322-4517

1. Abstract

This paper presents a methodology for evaluating reusable Ada software libraries. It identifies a set of criteria and analyzes how to apply them. Three estimated quantities and two metrics are defined. The quantities are the degrees of criterion satisfaction, importance, and certainty. One of the metrics is intuitive and in different incarnations is widely followed; the other is based on notions of fuzzy set theory and merits a larger following. The methodology is flexible and encourages adaptation to local requirements and capabilities.

KEYWORDS: Abstract Data Types, Code Complexity, Cost Effectiveness, Cost Model, Domain Analysis, Domain Coverage, Fault Tolerance, Generic Units, Object-Oriented Design, Package Cohesion, Performance, Portability, Reusability, Security, Software Components, Software Libraries, Software Metrics, Software Repository, Software Testing.

2. Introduction

In recent years, as budgets have shrunk and demands on software have grown, interest in software reuse has spread throughout the Ada community. With this interest has come recognition of the many challenging issues involved in implementing an effective software reuse program. [2,3] One of these issues is how to evaluate a reusable Ada software library for adoption in system development. In this paper, we provide a set of criteria for evaluating reusable Ada software libraries and a methodology that uses them. The methodology is flexible and encourages, indeed *requires*, you to customize or tailor it to your company goals, requirements, and capabilities.

The criteria may be divided into five broad areas: 1) Reliability, 2) Functionality, 3) Software Engineering, 4) Vendor Support, and 5) Cost Effectiveness. The criteria are

- Reliability,
- Popularity,
- Security,
- Software Testing,
- Functionality,
- Application Domain Coverage,
- Abstract Data Type Coverage,
- Mathematical Coverage,
- Graphics Coverage,
- Software Engineering,
- Customizability,
- Fault Tolerance,
- Code Simplicity,
- Package Cohesion,
- Code Portability,

- Platform Compatibility,
- Vendor Support,
- Documentation Quality,
- Performance,
- Enhancement Rate,
- Tool Support,
- Maintenance, and
- Cost Effectiveness.

For each criterion, three measures are defined: the criterion rating itself, its importance, and its certainty. All are estimates based on research into the reusable Ada library. Certainty is often ignored but just as significant as the other two.

To promote ready comparison between different libraries, two Reusable Ada Software Library Evaluation (RASLE) metrics are also defined. Calculation of one is quite intuitive, that of the other less so because it draws on concepts from fuzzy set theory. (If you would like a copy of an Ada program that computes these metrics given a set of values for the evaluative criteria, please send a self-addressed envelope and a 5 1/4 " high density diskette to the author.)

3. Criteria for Evaluation of Reusable Ada Software Libraries

In this section, we describe each criterion, explain its justification, and examine ways to measure it. After careful evaluation, each criterion is assigned an integer value in the suggested range 0..4. You can certainly use a different range of values, but more values requires a finer sense of discrimination than is likely regarding the application of the criteria to particular reusable libraries. The lower limit is 0 to simplify calculations.

Many of the criteria that make good reusable software make good software in general. However, the two do not always coincide – sometimes, in fact, they conflict. Further, even some of the criteria that enhance software reusability may conflict among themselves.

Few, if any, of the criteria listed are sharply defined. If they were, evaluation of reusable Ada libraries would be easy, but it's not. Some considerations in applying a criterion may easily be interpreted as falling under another criterion differently conceived. Indeed, grouping the considerations differently may construct a different set of criteria. To handle the problem, you must carefully keep in mind the considerations used to determine the satisfaction of each criterion.

Software Testing. Like the reliability of software in general, the reliability of a library depends on extensive test coverage. Many libraries provide a suite of tests used to test the software components. Ideally, during the evaluation process, you would like to run the tests yourself, examine the tests for coverage and validity, and add your own tests to the test suite. Do the tests adequately cover the range of valid and invalid input values? Do they let you test for a variety of sequences of inputs? Can you see the test results?

Popularity Here we read the literature and talk to associates whose judgment we respect. Have many other companies used the libary's components with satisfaction? Of course, a piece of popular software may not be what you want. In this regard, a healthy skepticism is a faithful friend. Nonetheless, the criterion is valuable because popularity is often justified. Note, too, that with reusable Ada software libraries, determining popularity may be difficult because many companies are new to offering reusable Ada software, and most companies (alas...) are still slow to reuse it.

The age of the company and its software library may themselves be considered in assessing satisfaction of this criterion. Mere age may not indicate widespread popularity, but it does suggest that some have found the company's products (even, perhaps, the software you're evaluating!) worthwhile.

Security or Library Integrity. In choosing software from a library, we trust that the delivered software is what its developers and distributors believe it to be. We would not be pleased to know, for instance, that the software had been accidentally or maliciously changed so that bugs were introduced or, worse, that a virus of some sort lurked within it. It is difficult to assess the satisfaction of this security criterion, especially before delivery as we'd like. In view of a number of disconcerting incidents that have been publicized in recent years, it appears that, other things being equal, accessibility to the library via a public network would tend to lower the satisfaction rating of this criterion. However, if you can later scan the delivered software with your trusted virus detection program, you may be able to raise the rating.

Domain Coverage. The extent to which the library implements the algorithms specific to your application domain may be crucial. In the Intelligence/Electonic Warfare (IEW) domain, for instance, Fast Fourier Transforms, Polar/Cartesian conversions, and azimuth determination algorithms, among others, will often be significant. A list of relevant algorithms should be constructed, if indeed a more thorough domain analysis is not attempted. Care should be exercised that areas of overlap among criteria not inflate the measures for them.

Abstract Data Type Coverage. Several reusable Ada libraries cover this area in varying degrees. Perhaps, a library that has small coverage may be exactly what you want. Discuss with your software engineers what your potential applications require: lists, queues, stacks, strings, trees, networks, or others? What operations will your applications need for a given data type? Are they among those in the library? If not, can you build them readily from those provided? Also, for the operations you want, are there different implementations based on the subprogram's usage of time and space? Watch potential overlap with the performance criterion below.

Mathematical Coverage. As with other coverage criteria, we must determine what percentage of functions needed are in the library under evaluation. Many common operations are available in several libraries. These include trigonometric functions, matrix operations, Gaussian linear equation solvers, numeric integration, statistics, Newton function solvers, differential equations, and special functions. Not all of these are found in a library even if it concentrates on mathematical routines in Ada. Different versions of the same functions varying by efficiency or accuracy should be considered.

Graphics Coverage. Consider whether your application must directly call low-level routines (draw lines, set pixels, draw circles, for instance) or higher level routines that provide icons, windows, menus, forms, etc. Different reusable Ada libraries focus on different levels.

For portability, note whether the libraries claim to implement graphics standards (whether formal or de facto) such as PHIGS and X-Windows. In this area more than others, coverage is dictated by the asssociated hardware (graphics board, terminal), compiler, and host and target machines. Some Ada graphics libraries may not compile under certain compilers or performance of the code may suffer.

Customization/Extensibility. No matter how good a library is, you may want to customize some routines. As a rough guide, the greater percentage of subprograms (or the packages containing them) that are generics, the better. For instantiation, can you define your own types (your own Complex number type, for instance) or must you use those supplied by the library packages themselves? How easily can you modify or enhance the subprograms? Do they use local variables or (more difficult, though often necessary) non-local ones accessed by many subprograms?

Fault Tolerance How well do the library routines use Ada's exception handling facility? Are exceptions defined where necessary or are the predefined Ada exceptions sufficient? Predefined exceptions may be sufficient for many mathematical routines, but with abstract data types, for instance, exceptions are often needed to handle things that can go wrong. Propagating a newly defined exception instead of a predefined one can more easily identify the source of a problem. Are handlers provided where necessary, or do they obstruct your application code from handling them?

Code Simplicity (Low Complexity) To modify reusable code, we do not want it to be complex. The maximum subprogram length should not exceed 100 lines or so (two or three pages), and the average length should be quite less. Further, are ifs, case statements, loops, blocks, or program units often nested to more than two or three levels? If McCabe, Halsted or other metrics are available, they may be helpful. Otherwise, you can study code samples, if possible. The greater the complexity, the lower the rating for this criterion.

Package Cohesion This feature makes it easier to find and integrate the code you need. Some facets may promote object-oriented design. Are subprograms that operate on the same class of objects grouped together in the same package? Do packages typically depend on (*with*) a small number of other packages? Are derived types used so that new classes of objects can inherit the operations of their parent types? Are related generic packages (e.g., operating on different kinds of matrices) nested in the same package? Nesting may be appropriate, despite potentially greater complexity.

Code Portability. Code portability is closely related to reusability, but it is not the same thing. If a large module of code (say, a CSCI) is portable, its components may not be reusable if they cannot be readily separated from their original application. (To some degree, it is also the opposite of the compatibility criterion.) This measure requires that there be little, if any, use of *implementation-dependent features from* Chapter 13, I/O, or elsewhere: address clauses, pragmas, unchecked conversion and deallocation, I/O FORM parameter, etc. In general, it also demands, for instance, frequent use of programmer-defined (not predefined) numeric types and constants, not literals, in the code. To a large extent, measuring the satisfaction of this criterion could be automated.

Platform Compatibility. We must note that the reusable code will work on our prospective system, not another one. Does the library use some routines (low-level math routines, for instance) written in FORTRAN, C, or another language? Do we have a compiler for it? Do the library subprograms compile under our compiler? (Compiler validation is no guarantee.) If so, do they run as efficiently on our system as they do on others? Different array storage layouts can affect execution speed.

Documentation Quality. Here we should consider the clarity, comprehensiveness, and organization of the documentation; the quality and placement of comments in the code; the informativeness of variable names; and the ease of access of specific topics. Are there indexes as well as tables of contents for each document? Are the limitations on the algorithms explained or referenced? Are diagrams (hierarchy charts, data flow diagrams, etc.) adequate? Is the documentation available electronically as well as in hard-copy? It is sometimes difficult to obtain an evaluation copy of the library's documentation, but some vendors will freely provide portions of the documentation or a complete set if a non-disclosure agreement is signed.

Performance Data. Is there performance data on the library routines as run on your machine? If not, is a suite of performance tests provided? Tests for functionality are common, but for performance measurement not. Can you sample the code and write your own tests? Especially with matrix manipulation routines, this may be important. There may also be some penalty from object-based techniques.

Enhancement Rate. Past enhancements may be a key to future positive developments. Is the vendor planning to add routines to the library, in the same or a different domain? Is a new version for another compiler/host/target in the works? But this may not be so important if the product is already more than what you want.

Tool Support. This may be necessary only when the library is large, or generics are deeply nested or have many parameters. Some tools may assist in finding the component you need, or in building it through instantiations. Are packages named informatively? Is the library organized? Some graphics libraries assist you in prototyping the man/machine interface for your application and then generate Ada code to integrate with your application.

Maintenance. We all need support now and then. Can you easily obtain all the documentation you need? Is a maintenance agreement provided? Is there a warranty of any kind? This is rare. When you call, can you readily speak with a knowledgeable technical person? Is there more than one person who can answer your questions?

Cost Effectiveness. Only a careful cost analysis can adequately answer this question. (See [5] for a thoughtful approach.) A preliminary assessment, however, is appropriate. What is the library's cost? Some have a fixed cost, others a renewable license fee. Prices may vary according to whether the license is for one CPU, all those at a site, or the entire company. To be optimistic, prices may drop as reuse catches on. (Little analysis has focused on what reusable software should cost. [1]) Can you purchase just what you need, or must you purchase a whole suite of packages?

What are maintenance costs?

4. *Two Measures for Evaluating Reusable Ada Libraries*

Associated with each criterion are two measures used to construct the reusable software library evaluation metrics. One is for criterion importance, the other for criterion certainty.

Importance is the degree of urgency you attach to the satisfaction of a criterion. The more you can do without satisfaction of the criterion, the less important it is. Importance is measured by an integer in the range 0..3.

The importance measures that you attach to criteria are not absolute. In many cases, they should vary according to several factors, notably the characteristics of the system being developed, the prospects for later porting to another hardware platform, the Software Engineering Environment in which development occurs, the calibre of your software development staff, and other factors.

Certainty is the measure of justifiable assurance or credence that the numeric measure you assign a specific criterion (whatever its value) is indeed accurate. By their very nature, the values assigned to the evaluative criteria are subject to error or inaccuracy because of misunderstanding, inadequacy of available information, etc. Sometimes, you can review code and documentation samples before purchase, if you sign a non-disclosure agreement.

To mark its probabilistic nature, certainty is given a value in the range 0..100. Roughly, we can expect that the more research effort spent on investigating a reusable library, the more certain we can be that our evaluations for criteria satisfaction are accurate.

The measures of importance and certainty should be determined initially and periodically revisited throughout the evaluation process. Just as your measures of certainty change as you learn more about the reusable libraries under review, so may your measures of importance as you learn more about the system(s) under development.

5. *Two Metrics for Evaluating Reusable Ada Libraries*

To determine the intuitive RASLE1 metric:

- Multiply each criterion measure by its associated importance measure
- Total up all the products so derived

This metric is similar to a statistical weighted average or expected value, but it is not divided by the total of the importance measures.

The RASLE2 metric uses comparable operations in fuzzy set theory, namely the union, intersection, and complement of two fuzzy sets. [4,6,7] The union of two fuzzy sets is the maximum of the values in the sets. The intersection is the minimum of the values. The complement of a value is the highest possible value minus the given value. Briefly, the steps are:

- Compute the union of each criterion measure with the complement of its importance.
- Determine the intersection of all the library's unions so derived.

The two RASLE metrics give comparable results. To ensure validity in your results, we recommend you use both.

To determine a certainty measure for the library as a whole, the individual certainty measures are treated like the criteria values. Together, either RASLE metric and its associated certainty metric (RASLEC1 or RASLEC2) form a composite tool for evaluating reusable libraries.

Like all metrics, the RASLE metrics for the libraries under consideration must be used carefully. The metrics provide a tool for your buy or build decision -- they do not determine it. The metrics of libraries offering comparable domain coverage should be compared, and cost analyses with those libraries having high ratings should be made.

6. *Summary and Conclusions*

The methodology presented here requires that you thoughtfully and carefully

- Determine the degrees of importance you attach to all the criteria
- Assess the degree to which each criterion is satisfied for each library under review
- Evaluate the uncertainty attaching to your ratings of each criterion for that library for each library under review
- Calculate the software metrics evaluating the reusable software libraries
- Compare the software metrics of two or more reusable libraries that share strong ratings in an area of coverage of special interest

Doing this well requires that you spend time, money, and research – elements to be included in the cost modelling that you perform for introducing and maintaining a software reuse program in your company. Yet, the promise of significant savings in the intermediate and long term makes the effort seem well worth it. A comparative cost analysis with the reusable libraries scoring highest in a given class, and made in light of a number of projected systems, should go far toward answering the question.

Finally, note that many of the criteria presented here may also be used to provide guideposts and standards to follow in writing reusable software afresh or in modifying current software to make it more reusable. That, however, is another topic.

7. *Acknowledgments*

This methodology has been used in support of the work of the U.S. Army Communication and Electronics Command (CECOM)'s Center for IEW (Intelligence/Electronic Warfare) Systems Working Group on Reusable Software. The writer would like to thank the members of this working group – especially Cenap Dada (Working Group Chair), Jim Iverson, Jerry Brown (all of CECOM), and Dan Berube of COMCON, Inc. – as well as John Medea and Bob Marchand (both of COMCON, Inc.) for their helpful discussions.

8. *References*

1. Aharonian, G. "Pricing a Reusable Software Package," *Defense Computing*, Nov/Dec, 1989.
2. Booch, G. *Software Components with Ada*, Benjamin/Cummings, Redwood City, CA, 1987.
3. Braun C., Goodenough, J., and Eanes, R. *Ada Reusability Guidelines*, Technical Report, SofTech, Inc., Waltham, MA, 1985.

4. Caudill, M. "Using Neural Nets, Part 2: Fuzzy Decisions," *AI Expert*, April, 1990.

5. Gaffney, J. "An Economic Foundation for Software Reuse", Software Productivity Consortium, VA, July, 1989.

6. Yager, R. "Concepts, Theory, and Techniques: A New Methodology for Ordinal Multiobjective Decisions Based on Fuzzy Sets," *Decision Sciences*, 12(4) October, 1981.

7. Zadeh, L. "Fuzzy Sets," *Information and Control*, 8, 1965.

9. About the Author

Stephen H. Levy is a computer scientist and president of Calculemics, Inc., a firm specializing in computer research, development, and training. His clients include U.S. Army CECOM's Center for Software Engineering, AT&T Bell Laboratories, SofTech, Inc., UNIX Pros, Inc., and others. He earned a Ph.D. in the philosophy of mathematics and logic at Fordham University, and is the author of *Ada: The Fortran Programmer's Companion*, Silicon Press/Prentice-Hall International, 1991. Dr. Levy's research interests include software engineering, programming languages, mathematical applications, and logic and reasoning.

FACILITATING REUSE IN A SOFTWARE ENGINEERING ENVIRONMENT

John C. Schettino, Jr.
Catherine S. Kozlowski

Contel Technology Center, Contel Federal Systems

Abstract: There are a number of problems with library-based reuse methods. Technical issues such as ease of creation, maintenance, and enrichment of the library, as well as the resistance of software engineers to reuse programs, impede the success of reuse efforts. Additional procedures and methods must be instituted in order for a reuse program to succeed. The value of the reuse program must be measurable. For these reasons, stand-alone reuse library systems provide only a partial solution to reuse efforts. The integration of a reuse library system into an Integrated Process Support Environment (IPSE) which provides process control is one method of providing a more complete solution. The information and process control provided in an IPSE complements the functions provided by reuse libraries, creating an environment which supports and encourages reuse.

What Is reuse?

There are a number of work products created during the lifecycle of a single software system. These work products traditionally are conceived and realized without regard for previous efforts which may partially or completely solve the problem at hand. In the distant past, all software and hardware development was performed this way. Every solution to a problem was a unique invention, different from all others. Contrast this early state of the art with today's hardware engineering environment, where designers are taught methods for selecting and combining reusable building blocks of components in order to design and implement new or improved systems. Building blocks have standard, documented interfaces, and are readily available from a number of sources. Designers are taught to consider existing building blocks when analyzing and specifying new systems, and generally do not implement brand-new components to complete a system. The creation of new building blocks has become a separate industry, with different skills required. Current methods of software design and implementation have incorporated a large amount of reuse into the process of analysis, design, and implementation of software systems, but not as much as in the development of hardware systems. Current "every day" reuse in software includes the use of operating systems, standard languages (and their supporting low-level libraries) and standard mathematics libraries. The widespread reuse of the work products of past efforts, even within an organization, is still an unrealized goal. These work products represent a large investment in effort and money, which could be used to speed future development of similar systems, improve quality by use of refined and tested building blocks, and free software developers from the constant re-implementation of solutions.

Reuse today

Large organizations have realized the value of reuse, and have attempted to implement procedures, create tools, and prescribe processes to encourage and increase reuse in their software development efforts. A common approach is to create a manual or computer based software component library, and a software development process which includes reuse activities. The library includes a classification system which identifies the types and uses of the components, and a method or tool for locating and retrieving desired components. Raytheon[1] and NEC[2] have successfully instituted reuse programs which address common business applications code within existing application domains. This limited approach to reuse simplifies the requirements of the classification and retrieval systems, since the domain is limited and somewhat static. These systems do not readily address the classification of new modules or domains which fall outside of the established classification hierarchy. Other organizations, such as Fujitsu[3] and the Army's RAPID Center[4], approach reuse more pragmatically, mandating that all software projects formally include the reuse organization within their development cycle. The reuse organization is modeled after a regular library, with a staff of systems analysts, software engineers, reuse experts, and domain experts required to support its functions. This represents a significant investment in reuse, and off-loads the problems of classification and retrieval to the library staff. Finally, GTE Data Services[5] has instituted a corporate-wide program, with the stated goal of developing a reuse culture. This program encompasses management procedures and tools which support reuse. Several groups support the activities of the software development lifecycle, including management, classification, maintenance, development, and reuser support. These groups represent a new culture and process which must be assimilated prior to implementing reuse in an organization.

The reuse problem

The approach described in this paper addresses the following three problem areas for library-based reuse of software lifecycle work products:

The creation, enrichment, and maintenance of a library

Library-based systems require support and ongoing maintenance. Manual systems will need support staff to maintain catalogs of components, and to assist in the selection and extraction of components from the library. All library systems require some form of classification for each work product in the system, and the update and maintenance of the work products in the library. Procedures must be implemented prescribing the use of the library, and the method of submitting new or enhanced work products into the library.

The resistance of software engineers to reuse programs

The use of a library of work products must not overly complicate the analysis, design, and implementation of software systems. Software engineers must be encouraged to perform reuse activities, and be directed to explore existing solutions prior to creating a new solution to the problem at hand.

Assessing the value and level of reuse in a project

Management must be able to assess the success of reuse efforts in a project. The specific amount of effort spent in attempting to reuse work products, the resulting amount of reuse in a given project, and the success of the project and the individuals within a project in reusing work products should be provided by the process and library systems.

The implementation of library based reuse systems poses several problems. An approach which is difficult to use will act as a disincentive to the primary users of the system. An approach which is difficult to administer will overly complicate and burden the process of software development. An approach which does not provide usage metrics will be difficult to assess. Even when these problems are addressed, a library system is only a partial solution. The library system must be readily available, easy and pleasant to use, and directly supported by the software development process.

Contel has approached the problem of library-based reuse in its Federal Systems Division through the creation of a library system which is integrated into a software engineering environment which directly supports and encourages reuse. This is accomplished by automating both the tools and the process for performing reuse activities within a central controlling environment for software development.

The Component Retrieval System

The Contel Technology Center (CTC) has created a series of prototypes of a Component Retrieval System (CRS) which use a faceted classification approach[6] to classify and retrieve reusable components. A component is any work product from the software system lifecycle, including algorithms, module designs, software modules, specifications, requirements, test plans, etc. The use of a faceted classification approach and domain partitioned collections address two major inhibitors of reuse: simplification of searching for particular components and assessment of the relevance of a reusable component to a particular application.

The goals of the current reuse library prototype

The current CRS prototype was designed to address the limitations of the current library systems in a number of areas. The implementation is based on the UNIX operating system and X Windows graphical user interface (GUI) technology. This technology provides a bit mapped, windowing environment with multiple concurrent processes, while providing a consistent and understandable user interface. It also provides simplified integration of the CRS into an Integrated Process Support Environment (IPSE). The use of a windowing GUI allowed for a consistent user interface design, with functions grouped logically into different windows. This improves the usability of the system.

Simplification of the creation and administration of a library. The CRS is designed such that all features needed by a librarian are integrated into the system, and are simply unavailable to non-librarian users. All of the librarian's tasks, such as the classification of new components, are performed within the CRS windowed environment. There is no need to understand the underlying DBMS/repository, or to learn any database query language.

Improved usability. The CRS presents a consistent window and menu based user interface. Each window contains a menu bar at the top showing all available groups of functions, with drop-down menus for each activity showing individual functions. Unavailable functions are "grayed-out" indicating that they are currently inappropriate. Wherever possible, choices are presented as scrolling lists that the user may use to select desired values. Consistency checks and error dialogs are provided to prevent user errors. Context sensitive help is available for every user action. Users may customize the layout of windows, and establish default values for items such as work product extraction paths, printers to use for output, and search parameters.

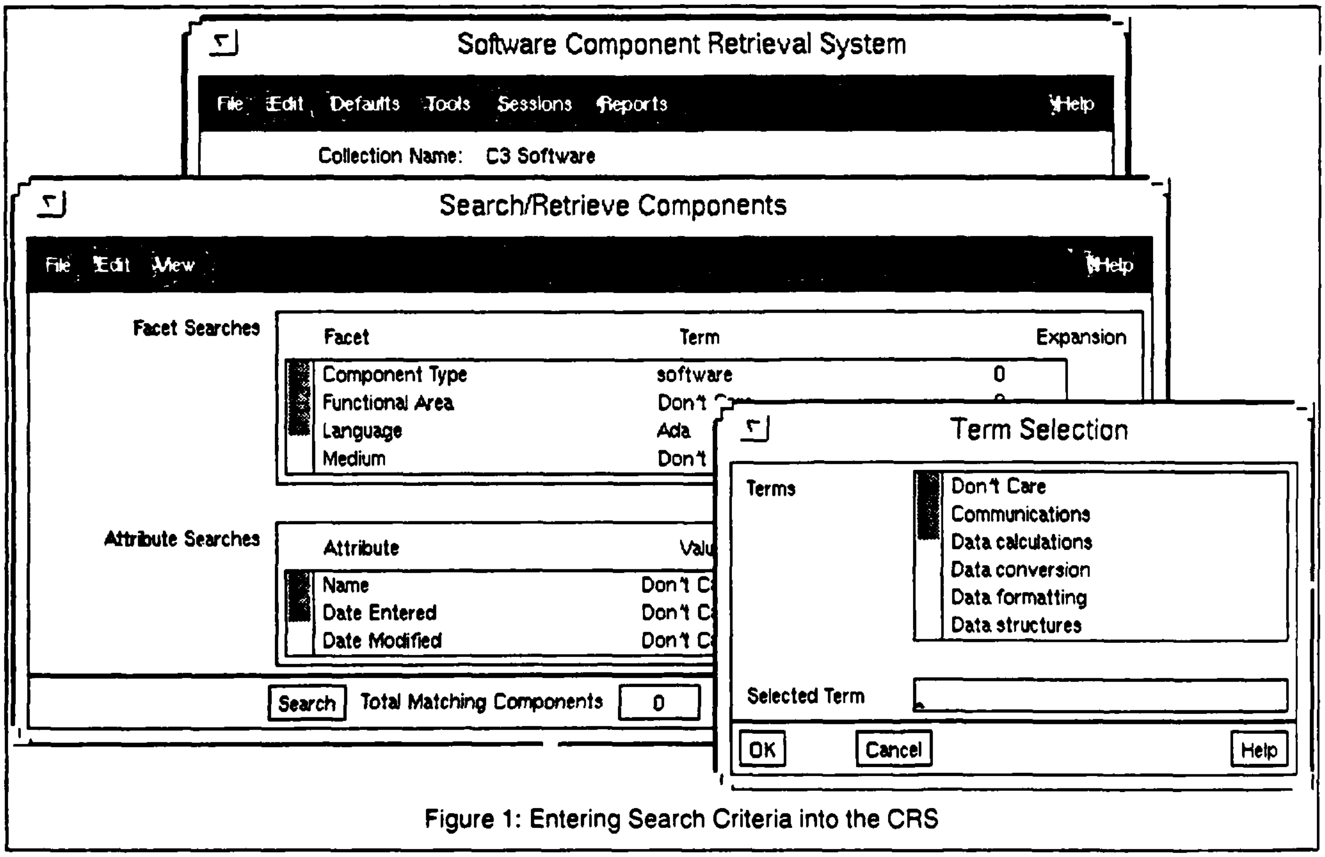
Improved Availability. The CRS is implemented such that a number of users may access the centralized repository simultaneously without degrading response times of library queries. The entire tool can be closed down into a desktop icon and left active within the IPSE while the user performs other tasks. It may then be accessed again without delay when needed.

Detailed usage metrics. The CRS tracks a number of items within the system and produces a number of metrics on usage. Each user may be assigned to a project. If this is done, the CRS will provide project-wide reuse metrics including user sessions, number of searches, number of matches, number of extractions, and other information. The tool provides metrics on the use of the vocabulary for searches, which can assist the librarian in fine-tuning the classification system.

A tour of the CRS

Getting started: When a software engineer specifies a module's functionality and the data it needs to manipulate, and then wishes to locate a pre-existing module for reuse, he utilizes the CRS to find existing modules in the component library which may satisfy those needs.

Searching: The process of searching for a reusable component in the CRS involves first selecting the collection, or domain, and then specifying the characteristics of the desired module using the faceted classification system. Figure 1 shows an example where the user is about to select a value for the Functional Area facet for use in a search. The Main window (mostly obscured) shows information on the selected collection. The Search/Retrieve window contains the search criteria, and the Term Selection window (the topmost window) has just popped-up in response to a mouse click in the Term column next to the Functional Area facet.



Collection Name: C3 Software

Facet Searches

Facet	Term	Expansion
Component Type	software	0
Functional Area	Don't	
Language	Ada	
Medium	Don't	

Attribute Searches

Attribute	Valu
Name	Don't C
Date Entered	Don't C
Date Modified	Don't C

Total Matching Components 0

Terms

- Don't Care
- Communications
- Data calculations
- Data conversion
- Data formatting
- Data structures

Selected Term

Figure 1: Entering Search Criteria into the CRS

Fortunately, it is much easier to perform these activities than it is to explain them!

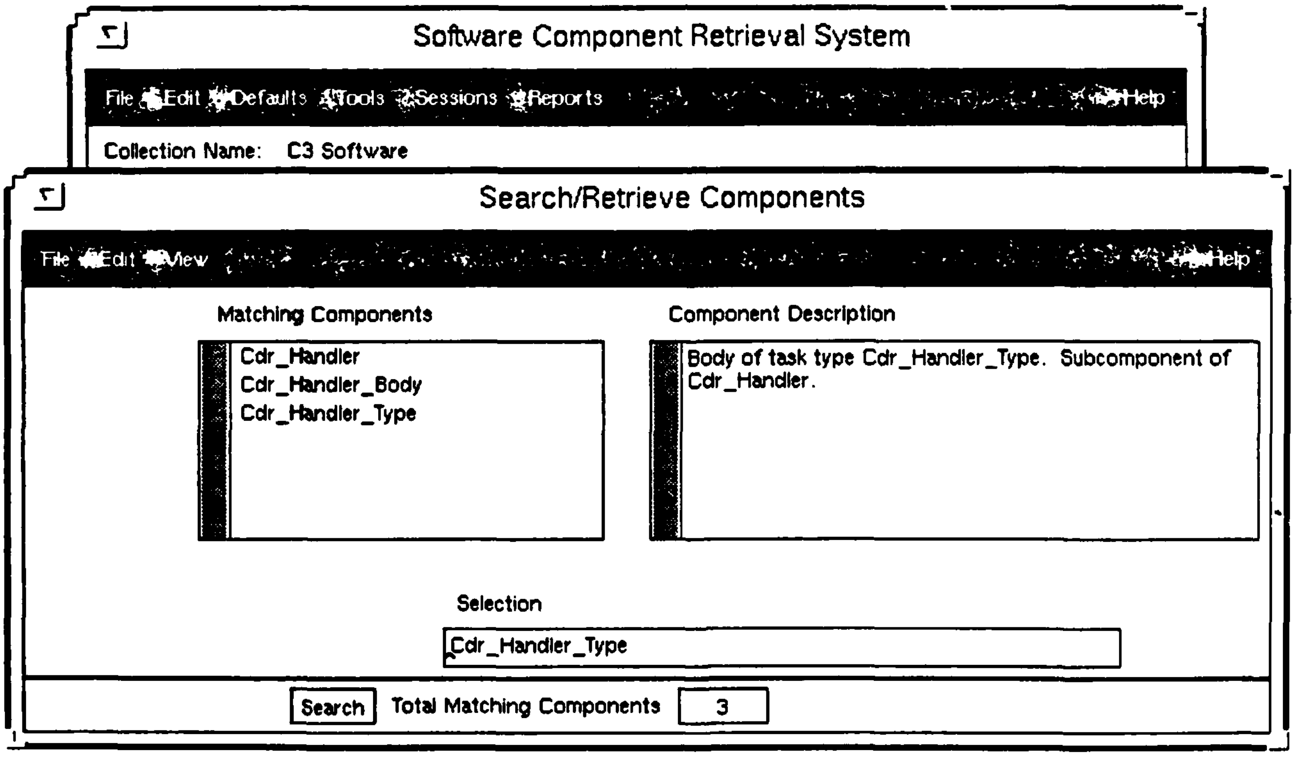
Executing a search: At any time during search criteria entry, the user may get a count of the number of matching components based on the current search criteria by pressing the Search button. This will update the Total Matching Components field, without actually retrieving any components. In this way the user may start with a more general search, and add additional constraining facets until the count of matching components is small enough to browse through.

Selecting a matching component for inspection: The CRS will display a list of all candidate modules which exist within a collection that match the desired characteristics, and then allow tne software engineer to inspect any of the matc.hing components for possible reuse. Figure 2 shows an example where the user has selected the component named Cdr_Handler_Type. The Search/Retrieve window contains the matching components from the search, and the selected component's description is displayed.

Inspecting and Extracting components: The CRS will display a component detail window which allows for detailed inspection of a matching component. This window can display four different views of a component. As shown in Figure 3, the four views provide sufficient information for the software engineer to decide whether or not to extract and reuse the component. The first view shows the full faceted classification for the selected component. A second view displays the component's description and a count of the number of inspections and extractions performed on it, as well as the date the component was entered into the system and the date it was last accessed. The third view of the component shows any relationships and/or dependencies this component has with other components in the collection. The fourth view is of the component itself. If the software engineer decides to extract the component, the CRS will make a copy of the component (and optionally, all of its related and/or dependent components) into the user selected directory.

Component evolution: If a component is extracted from the database and is then subsequently adapted for the particular task, it would be a candidate for inclusion into the CRS collection as a new component which is related to the original. If a performance enhancement or error correction is performed, then the component might either replace the existing version or be placed into the CRS as a related component.

Collection enrichment: When a software engineer does not find an existing component in the CRS, the knowledge of the collection and classification of the newly created module, gathered from the searches



Collection Name: C3 Software

Matching Components

Cdr_Handler
Cdr_Handler_Body
Cdr_Handler_Type

Component Description

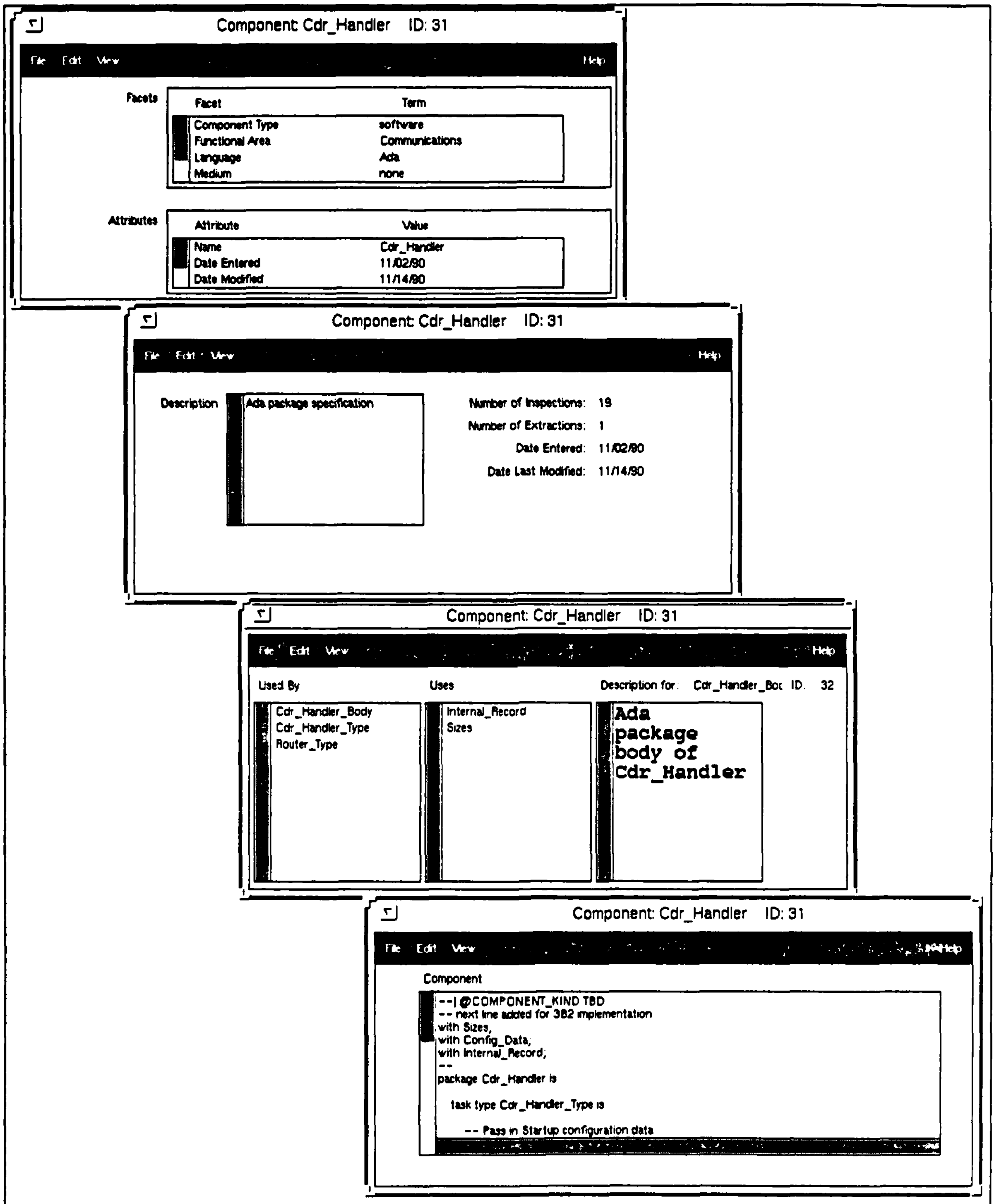
Body of task type Cdr_Handler_Type. Subcomponent of Cdr_Handler.

Selection

Cdr_Handler_Type

Total Matching Components 3

Figure 2: Selecting a matching component for inspection



Component Cdr_Handler ID: 31

Facets

Facet	Term
Component Type	software
Functional Area	Communications
Language	Ada
Medium	none

Attributes

Attribute	Value
Name	Cdr_Handler
Date Entered	11/02/90
Date Modified	11/14/90

Component Cdr_Handler ID: 31

Description: Ada package specification

Number of Inspections: 19

Number of Extractions: 1

Date Entered: 11/02/90

Date Last Modified: 11/14/90

Component: Cdr_Handler ID: 31

Used By	Uses	Description for: Cdr_Handler_Boc ID: 32
Cdr_Handler_Body Cdr_Handler_Type Router_Type	Internal_Record Sizes	Ada package body of Cdr_Handler

Component: Cdr_Handler ID: 31

Component

```
--| @COMPONENT_KIND TBD
-- next line added for 382 implementation
with Sizes,
with Config_Data;
with Internal_Record;
--
package Cdr_Handler is

   task type Cdr_Handler_Type is

      -- Pass in Startup configuration data
```

Figure 3: Views of a component

attempted by the software engineer, could be used when adding it to the CRS. The component should be designed for reuse using language specific reusability guidelines. A number of guidelines exist for the Ada[7], C and COBOL[5] languages.

Automated classification of new components is not currently feasible, but as discussed above, the process of determining whether or not a component exists provides valuable information about a component created to fill a need not currently addressed in the reuse library. A method which captures this information can be provided through integration of the CRS into an IPSE which has full control of all objects within the environment.

The collection and evaluation of usage data within the CRS provides insights into the classification system's accuracy in specifying the components contained within a collection, as well as usage patterns of development projects and individual software engineers. Coordination of this data with the project-wide data provided by an IPSE can provide metrics on reuse levels within a project, cost savings, and reuse library enhancement on a per-project basis, and can aid in the effort estimation of similar projects based on past reuse levels.

The Process Enhancement Program

Contel Federal Systems (CFS) Government Systems Group established the Process Enhancement Program (PEP) in January of 1989. The PEP is Contel's Software Engineering Process Group (SEPG), and is chartered to systematically improve CFS' ability to win, and to perform successfully on, software-intensive Government systems contracts. The PEP is addressing the following four areas where improvement is needed: Policies, Procedures and Standards, Metrics and Estimation, Training, and Tools and Environment. One of the advantages of this effort is that the PEP began the improvement process with no pre-conceived notions of either what the process would look like or how much it would cost to complete.

The Tools and Environment task area of the PEP is responsible for defining and constructing a highly automated, fully Integrated Process Support Environment (IPSE) for system design and development. The environment must be capable of supporting a wide variety of different types of users (managers, analysts, developers, testers, etc.) in a distributed network environment.

PEP has two major goals for the IPSE. The first goal reflects the major reason why an organization would undertake building or acquiring an IPSE: increasing product quality. The second goal is unique to the overall approach of the PEP; i.e., the IPSE is being designed and engineered to automatically enforce the relevant procedures, standards, and lifecycle methodology for each individual project that uses the IPSE.

Adding value through Integration

The CTC and PEP are now jointly working on integrating the CRS into the PEP IPSE. This integration is intended to provide the engineers using the IPSE fast, easy and convenient access to the CRS database, while at the same time providing the CRS database with immediate access to baselined and approved project data. This integration provides the ability to have the IPSE serve as the single point of focus during the entire project development effort. Not only are the tools available for all of the activities of new software development, but now the ability to access a reusable component database has also been incorporated.

In order to take full advantage of the CRS integration, the PEP is adjusting the project lifecycle or process definition steps to incorporate reuse. This is intended to establish the culture that an engineer should first investigate the CRS domain that matches his project before embarking on a new development effort. The CRS to IPSE integration method will ensure that software engineers are directed to the CRS at multiple points in the development lifecycle, including requirements analysis, functional specification, design, and implementation. This facilitates reuse by providing several opportunities to locate existing work products throughout the lifecycle, and assures that development of new software to satisfy a requirement cannot take place until a search of the proper domain's reuse library has taken place.

Since future releases of the IPSE are intended to centrally manage each object within the environment, the integration of the CRS into the IPSE will utilize this control to communicate the status of a component as *reused* if it was extracted, or as *new* if no reusable component is located. For a *reused* component, information relating to the existing classification can be associated with the component while in the IPSE. For a *new* component, information captured during CRS searches for the component can be associated with the newly created component in the IPSE. In both cases, the information can be used as a classification aid when the component is completed and returned to the CRS. Additionally, the usage data captured from the CRS will be integrated into the IPSE data to enhance the analysis of project level metrics.

Conclusion

There are a number of problems with library-based reuse methods for storing and retrieving software lifecycle work products. Technical issues such as ease of creation, maintenance, and enrichment of the library, and availability of the library, as well as the resistance of software engineers to reuse programs, impede the success of reuse efforts in large organizations. Additional procedures and methods must be instituted in order for a reuse program to succeed. The value of the reuse program must be measurable, in order to assess its positive impact on the software development process. For these reasons, stand-alone reuse library systems provide only a partial solution to reuse efforts. The integration of a reuse library system into an IPSE which provides process control and project metrics is one method of creating a more complete solution. The information and process control in an IPSE complements the functionality of stand-alone reuse libraries, providing an integrated environment which supports and encourages the reuse of software engineering work products.

References

1. Lanergan, R.G. and Poynton, B.A., "Reusable Code: The Application Development Technique for the Future," Proceedings of the IBM SHARE/GUIDE Software Symposium, October, 1979.

2. NEC, Inc., personal communication with R. Prieto-Diaz, June, 1987.

3. Fujitsu, Inc., personal communication with R. Prieto-Diaz, June, 1987.

4. Guerrieri, E. "Searching for Reusable Software Components with the RAPID Center Library System," Proceedings, Sixth National Conference on Ada Technology, March, 1988.

5. Swanson, M. and Curry, S. "Results of an Asset Engineering Program: Predicting the Impact of Software Reuse," Proceedings of the National Conference on Software Reusability and Portability, September, 1987.

6. Prieto-Diaz, R. and Freeman, P., "Classifying Software for Reusability," IEEE Software, January 1987, pp. 6-17.

7. Braun, C.L., Goodenough, J.B., and Eanes, R.S., "Ada Reusability Guidelines," Technical Report 3285-2-208/2, SofTech, Inc., April 1985.

John C. Schettino, Jr. is a Senior Member of the Technical Staff at the Contel Technology Center (CTC). His interests include software reuse, information retrieval systems, and domain analysis. He received the BS in computer science from the University of Maryland, and the MS in computer science from George Mason University, where he is currently studying for the Ph.D. in information technology.

Catherine S. Kozlowski is the Director of the Process Enhancement Program (PEP) for Contel Federal System's Government Systems Group. As the Director, Ms. Kozlowski is tasked with providing both management and technical direction to the program. She has sixteen years experience in the analysis, design and development of large scale, distributed computer and communications systems. She was awarded the BS in physics, and has pursued graduate work in both nuclear engineering and computer science.

Questions about this paper can be addressed to the authors at the Contel Technology Center, 15000 Conference Center Drive, P.O. Box 10814, Chantilly, VA 22021-3808; (703) 818-4156

MEASURING ADA DESIGN TO PREDICT MAINTAINABILITY

Wei Li Sallie Henry Calvin Selig

Computer Science Department, Virginia Tech

Summary

This paper discusses the concepts of code metrics and design metrics. Several software complexity metrics are used in an empirical study. They include lines-of-code, Halstead's N, E, and V, McCabe's cyclomatic complexity,and Henry-Kafura's information flow. The metrics are collected from both the PDL Design and the resulting source code. The correlations between the metrics collected from the PDL design and the source code are given at different refinement levels. The correlations show that the information flow metric is independent of the refinement level, while the code metrics are dependent on the refinement level. The study result shows that the prediction of the source code complexity metrics may be possible from the PDL design. Based on the results of the study and several previous studies, a way to develop the methodology which predict the maintainability of software is presented. The statistical model is also discussed.

I. Introduction

The basic problem of the software crisis is that the development of large-scale software is out of control. In the last three decades, the research and application of software engineering developed many models which attempt to bring software development under control. Among them is the software life cycle model which analyzes the software development process by dividing it into phases. Although the software life cycle model provides certain disciplined ways of developing software, it does not provide a means of quantitatively measuring the product or process. To measure the software development process or product software metrics should be incorporated into the software life cycle. Software metrics are measures of certain characteristics of a development project. "You cannot control what you cannot measure" [1]. This fundamental reality underlies the importance of software metrics. From the software engineer or manager's perspective, metrics can represent a tool that, when properly used, enhances management control over the development process and product quality. When applying metrics the focus of the metrics is to achieve project-specific results. In general, the earlier in the life cycle when metrics are applied, the more control there is on the quality of the product [2].

This paper presents 1) the results of a research effort where software metrics were applied to the design; and 2) a methodology for using software metrics throughout any software development environment where the software development is divided into phases. Section II describes the software metrics and background studies. The advantage of Ada as a programming design language is also discussed. Section III presents the results of a study using metrics collected from the design using an Ada-like programming design language, to predict the source code quality. Section IV presents a methodology using software metrics throughout the software development environment. Section V is the conclusion.

II. Software Metrics

There are two types of metrics: qualitative and quantitative. Only quantitative metrics are of interest in this study. Software complexity metrics can be grouped into two categories *code metrics* and *design metrics*. Code metrics measure some features of source code. Design metrics attempt to measure the logic and interconnectivity of the system components. Hybrid metrics combine one or more code metrics with one or more structure metrics.

Code metrics are those that measure some attributes of the code such as length, number of control statements, and number of tokens, etc. Code metrics produce "counts" of some feature of the source code. The code metrics used in this study are : Lines of Code [3], Halstead's Software Science (N, V, E) [4], and McCabe's Cyclomatic Complexity [5]. The LOC (Lines of Code) is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. Halstead's N is the length of code, which is the sum of total occurrences of all operators and total occurrences of all operands. V, is the vocabulary of program which is the sum of number of unique operators and number of unique operands. E is the mental effort required to understand the program. McCabe's Cyclomatic Complexity is the number of basic paths through a program. It can be computed by counting the number of simple conditions within each decision statement and adding one.

Design metrics emphasize the overall system hierarchy and the interconnectivity of system components. The design metrics can be further divided into *structure metrics* and *hybrid metrics*. Structure metrics attempt to measure the logic and control flow of a program. These metrics take into account the interconnectivity among program modules that code metrics ignore. Interaction among modules (through shared data structure and procedure or package invocations) contributes to the overall complexity of the modules. The structure metrics that are incorporated in this study are Henry-Kafura Information Flow metrics [6] and McClure Invocation Complexity [7]. The Information Flow metric is based on the information flow connections between a module and its environment. McClure's Invocation Complexity is a function of the number of possible execution paths in the program and the difficulty of determining the paths for an arbitrary set of input data. Hybrid metrics take into account both the complexity contributed by system component (module) itself and by the interconnectivity of the system components. Hybrid metrics are a combination of one or more code metrics with one or more structure metrics. The hybrid metrics that this study incorporates are the hybrid form of Henry-Kafura Information Flow Complexity and Woodfield's Review Complexity [8]. Woodfield's Review Complexity attempts to measure effort in terms of the time required to understand a module. Although both code and design metrics result in numbers that somehow represent the "goodness" of a program, it has been shown that the two types of metrics are measuring different features of the software [9,10].

There are several studies the in metrics area which have encouraging results. In a study conducted by Henry and Selig [10], the software complexity metrics are collected from the design written in an Ada-like PDL, the metrics are then used to predict the source code quality. Details of the results from this study are given in Section III. There are two project-specific objectives in the study. One is to apply metrics to design. The other is to see how meaningful the design measurement is. Their results agree with Rombach's result . In Rombach's study, he finds that structure metrics are more useful in predicting the maintainability of a software system, and that most of the important structural decisions had been made irreversibly by the end of architectural design [11]. In another study conducted by Steve Wake and Sallie Henry [12], the metrics collected from the source code are used to predict the maintainability.

All of the above studies in metrics have encouraged us to conduct an experiment to develop a methodology which incorporates the software metrics into the software life cycle. In the methodology, metrics are collected from the design written in a PDL (Programming Design Language). The maintenance history of the code is collected. Then multiple linear regression is performed to get the prediction equation for the maintainability. This prediction equation could be used in the development environment as a quality control tool for either the manager or the system designer. Ada is recommended as the programming design language because 1) Ada provides a uniform, black-box view of different types of program modules; 2) Ada supports top-down refinement of nested modules in program test; 3) Ada supports the postponement of commitment to the target hardware configuration; 4) Ada supports information hiding through packaging; 5) Ada supports the separation of specifications and bodies of packages and tasks; 6) Ada supports a rendezvous mechanism for intertask communication. The tool to collect metrics automatically from design is also very important in applying metrics in a development environment. Such a tool has already been built as a front end in a Metric Analyzer in Virginia Tech [12,13].

III. Measuring The PDL Design

Over the past several years, Ada-like PDL designs and the resultant Pascal source code have been collected from undergraduate senior-level software engineering courses at both Virginia Tech and the University of Wisconsin-LaCrosse. The project-oriented class is designed to teach students the basics of software engineering. The goal of this course is to expose students to the experience of non-trivial program development in a "real world" environment where designing and implementing a project is not a single-person task. To this end, the class is divided into teams. Each team is responsible for designing a system that upon completion will be from three to five thousand lines of source code. This design includes hierarchy charts and module specifications written in an Ada-like PDL. After a team has completed the specifications for their project, they "hire" classmates to implement the modules in Pascal. Finally, the design team must integrate the modules into the completed program. The completed projects that have been finished by the classes are varied; games, development tools, and inventory systems. It is impossible to monitor all of the designs of all of the students; in fact, the students alone decide on "English-like" specifications, "code-like" specifications, or somewhere in-between.

After the PDL and Pascal code have been processed by the analyzer and metrics have been generated, procedures must be combined into modules. This is necessary because there must be a one-to-one correspondence between the design and source. The method used is simple. A single PDL procedure may be a definition of the function performed by several Pascal procedures. In order to equate the design and source, it is necessary to add the complexities of each of the Pascal routines. In this way, the design complexity is directly comparable to the source complexity on a functional level.There were 981 modules to analyze from 27 projects.

The simple use of correlations on a project-by-project basis is not very informative as far as the ability to predict source code complexity is concerned. Many of the design teams typify a problem in software development. Given a choice between doing a job well and doing the same job with as little effort as possible on their part, they choose the latter. This problem reinforces the belief that the specification phase of a project must be closely monitored to ensure that a useful design is the end result. Since the results of the correlations on a project-by-project basis give so little useful information, it is informative to look at the data as a single entity as opposed to 27 different segments. It is hoped that by doing so, the projects that are not well specified will be offset by those that are. It is also desirable to examine the data as a whole in order to develop predictors for each metric. Preliminary results indicate that the code metrics are strongly dependent on the level of the refinement while the structure metrics are independent of refinement level. This agrees with Rombach's result from another experiment [11]. A highly refined module would contain code-like specifications, while a low refinement level indicates the predominance of natural language specifications. To determine the validity of this hypothesis, the routines are divided into three categories; low, average, and high levels of refinement. Each level is analyzed individually, where the analysis consists of (1) correlations to determine the overall trends of the data, and (2) simple linear regression analysis to obtain the predictors.

As mentioned above, it is informative to examine the metric values based on the level of refinement of the specifications. In order to accomplish this purpose, it is necessary to determine the refinement level for each routine to be examined. It seems reasonable that as the refinement level increases, the length of the module also increases, but more importantly, the number of control structures will increase as well.

Table 1 shows the correlation of design metrics and the code metrics.

Table 1. Metric correlations by refinement level

Level	Lines of code	Halstead N	V	E	Cyclomatic complexity	Information flow
Low	0.610	0.512	0.552	0.211	0.405	0.903
Middle	0.701	0.731	0.756	0.577	0.867	0.908
High	0.810	0.630	0.807	0.706	0.902	0.792

The analysis of results in Table 1 shows that 1) Structure metrics required only a low level of refinement to be effective, but 2) Code metrics required at least a moderate amount of detail. The Information Flow metric appears to be independent of refinement level. The code metrics reacted as predicted. They do not perform well at a low level of refinement, and their correlations increase as level of refinement becomes greater. An interesting result is that the metrics perform quite well at even a middle refinement level. This indicates that after a minimum amount of detail is included in the specifications, the code metrics become useful measures.

Figures 1 and 2 show plots of the lines-of-code metric and the information-flow metric respectively, at a low level of refinement. Figure 3 shows the lines-of-code metric at a high refinement level, indicating the improved prediction. Figure 4 shows the information-flow metric at a high refinement level, demonstrating that the metric does equally well regardless of refinement level.

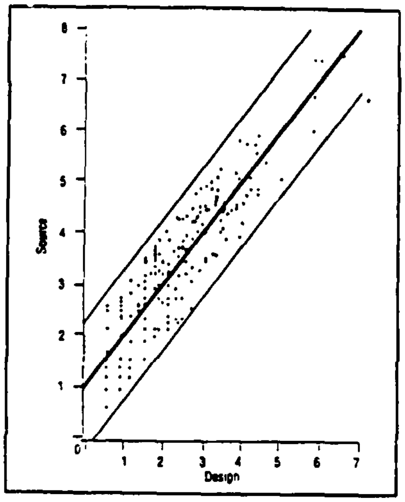


Figure 2. Low-refinement regression and 95%-confidence lines for information-flow metric

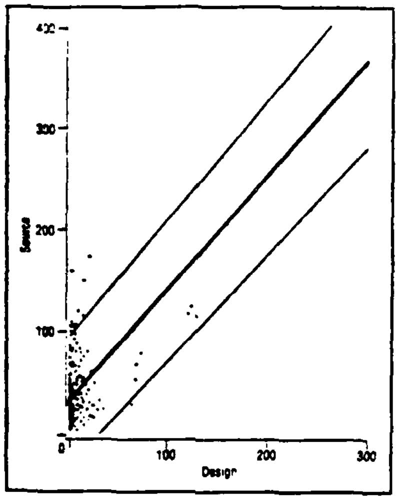


Figure 1. Low-refinement regression and 95%-confidence lines for the lines-of-code metric

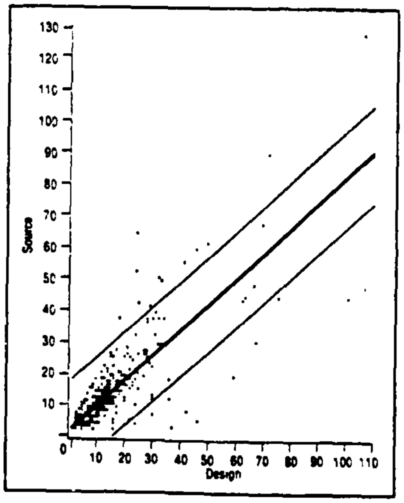


Figure 3. High-refinement regression and 95%-confidence lines for the lines-of-code metric

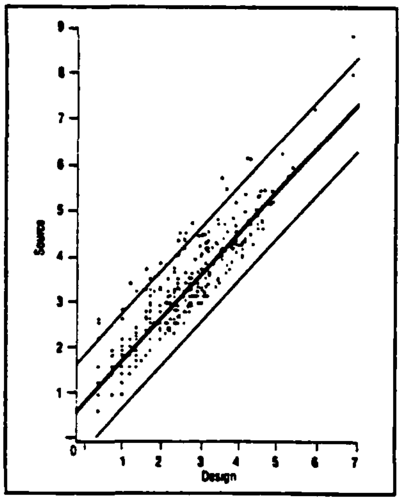


Figure 4. High-refinement regression and 95%-confidence lines for information-flow metric

Table 2 contains the inter-metric correlations for the Pascal code, verifying earlier studies. The data indicates that there is a high degree of correlation between code metrics. However, comparing the code metrics with the structure metric (information-flow) shows that the results are almost zero, indicating that the code and structure metrics are measuring different aspects of the source code. The inter-metric correlations, with respect to the design, show the same relationships as those using Pascal, and are given in Table 3. This is a desirable result since it indicates a consistency of measurement when comparing the designs to the resultant source code. It also lends credence to the usefulness of performing complexity measures at design time.

Table 2. Intermetric correlations for the Pascal code

Metric	Lines of code	Halstead *N*	*V*	*E*	Cyclomatic complexity
Halstead *N*	0.893	—	—	—	—
Halstead *V*	0.885	0.989	—	—	—
Halstead *E*	0.521	0.749	0.711	—	—
Cyclomatic complexity	0.629	0.776	0.781	0.492	—
Information flow	0.044	0.029	0.036	0.005	0.019

Table 3. Intermetric correlations for the PDL design

Metric	Lines of code	Halstead *N*	*V*	*E*	Cyclomatic complexity
Halstead *N*	0.894	—	—	—	—
Halstead *V*	0.894	0.988	—	—	—
Halstead *E*	0.465	0.725	0.681	—	—
Cyclomatic complexity	0.541	0.702	0.656	0.622	—
Information flow	0.260	0.208	0.249	0.039	0.039

This research shows three things that will extend the metrics utility and application : 1) structure and hybrid metrics are extremely useful at design time. (Although not treated separately from structure metrics, this study did include hybrid metrics, which were structure metrics with a small component of code metrics); 2) automatic generation of metrics for design specifications is not only possible with an analyzer similar tc the one used for this research, but it is also desirable; and 3) using the method for generating prediction equations, coupled with intimate knowledge of a detailed design, a designer can determine a specification's refinement level and determine the resulting source code's complexity.

IV. The Development of The Methodology

Several studies show the possibility of measuring the design with existing metrics [10,11,14]. Henry and Wake's study [15] shows the possibility of predicting maintainability using the metrics collected from source code. All of these previous results motivate us to conduct an experiment of applying software metrics throughout the software life cycle.

Several factors are considered in designing the experiment. The first factor is the selection of which metrics to use in the research. Among all the available software quality metrics, only those which are quantitative and automatable are used in this experiment. The second factor is when in the software development process to collect these metrics. Metrics collected from the code are successful in indicating error prone software components, uncovering difficult modules [9], and predicting the maintenance work required [15]. But metrics collected from code are considered too late to have significant influence on the redesign effort and the reduction of the overall budget. Collecting metrics earlier in the development process better guides the redesign effort and improves the quality of the end product, which in turn, reduces the overall budget [16]. In this experiment, metrics are collected from both the design and the source code. The third factor is the validation of the metrics. This depends on how the metric values are to be interpreted and used. In this experiment, the project-specific goal is to use the metrics to predict maintainability and guide the :design effort, therefore the successful validation of the metrics collected from design depends on how much of the maintainability can be predicted by the metrics. Therefore, the error history data of the end product is collected. The metrics collected from the design will then be used to predict the maintainability of the software product. With the metrics collected from the code for the same design, the results from previous studies will be verified.

Statistical Model

The use of the statistical model and the statistical analysis are the keys to the successful development of the methodology. Wake and Henry [15] indicated in their study that performing correlations between the error data and the individual metrics would not give satisfactory results since they do not feel that any one metric will perform equally well in any environment. Therefore, to determine the best set of metrics for the given environment, a regression model would give more information about the relationship of the error data and the design metrics. The previous studies have found that metrics in different categories measure different aspects of the software. Therefore, metrics from different categories together should predict maintainability better than those from any single category. In this experiment, multiple linear regression is used to predict maintainability from metrics collected at design time. In deciding the independent variables (metrics) that should be included in the model, the inter-metric correlation table shown in section III is considered. In order to avoid 'underfit' problem which might occur in regression when too few predictors are used, we include all the metrics collected using the Metric Analyzer tool developed in Virginia Tech [12]. In the multiple linear regression model, different metric values are used as independent variables, and the number of errors as the dependent variable. Multiple linear regression will be performed at procedure and system level. The multiple linear regression model for the experiment is defined as $Y=\beta_0+\beta_1*X_1+\beta_2*X_2+\beta_3*X_3+\beta_4*X_4+\beta_5*X_5+\beta_6*X_6+\beta_7*X_7+\beta_8*X_8+e$.

where Y is the dependent variable representing the number of errors. X_1 through X_8 are independent variables representing eight different metric values. The e is the error term in the regression. The eight independent variables are defined as following:

X_1 : Lines of Code.
X_2 : Halstead N.
X_3 : Halstead E.
X_4 : Halstead V.
X_5 : McCabe's Cyclomatic Complexity.
X_6 : Henry-Kafura Information Flow Complexity.
X_7 : McClure Invocation Complexity
X_8 : Woodfield's Review Complexity.

Statistical Analysis

The statistical analysis has five steps: dividing the data, significance test on the full model, variable selection, cross validation, and best prediction equation.

Step 1: Dividing the data

Randomly divide the data into two equal groups, group A and group B. Group A is used in steps 2 and 3. Dividing data into two groups can avoid the 'overfit' problem in regression. Whenever variable selection routine is used, there is a possibility that the predicting equation is tailored just to fit the sample data. To avoid this problem, the data is divided into two groups. Only one group is used to perform the regression. When the regression turns out to be significant, another group of data is used to verify the equation. And finally, the whole set of data is used to perform the multiple linear regression again to get the best predicting equation.

Step 2: *Significant test on the full model*

Using the F-test at 5% level to see if any prediction from the model is possible. The null hypothesis used is : H0: $\beta_1=\beta_2=\beta_3=\beta_4=\beta_5=\beta_6=\beta_7=\beta_8=0$. The goal is to reject the null hypothesis which means that some prediction is possible.

Step 3: Variable Selection

If from step 2 some prediction is possible, we would like to see if a subset of independent variables could be picked which does essentially as good of a job as the whole set of independent variables. Based on the previous studies, some metrics correlate very high with some other metrics. Since multicollinearity may exist among code metrics, the partial sum of squares of any code metrics given the rest of the code metrics may be near zero. This indicates that there might be some redundant predictors in the model. Therefore, not all of the code metrics may be useful in predicting the error. A subset of all the metrics will be decided in the final prediction equation. There are three different variable selection procedures. They are forward, backward and stepwise selection. There are four statistics that can be used to examine the quality of the prediction: R2, adjusted-R2, MSE (mean square error), and C(p). R2 is the amount of total variability accounted for by the regression in the sample. It increases when adding more predictors into the model. The adjusted-R2 is the estimated amount of total variability accounted for by the regression in the population. It is adjusted for the number of predictors in the regression model so that it may or may not increase as more predictors are included in the model. The adjusted-R2 is always less than R2. They are both within the range of 0 and 1. The MSE is the measurement of the error variance in the regression. It does not give more information than the adjusted-R2. It is possible that the regression model is biased (underfit) due to too few predictors in the model, or overfit due to redundant predictors The quality predictor takes into account of both possibilities. A small C(p) value should indicate that the regression model does not have a serious 'underfit' or 'overfit' problem. In this experiment, adjusted-R2, and C(p) are considered as the quality judgment for the regression equation.

Step 4: Cross Validation

The cross validation procedure is used to determine if the prediction equation is tailored too much for the sample data used in performing multiple linear regression. Any variable selection routine is using the sample to help determine the model. Thus it is entirely possible that we will tailor the model too precisely to the data at hand. In this event the model will look great for our sample but may not work at all in the population. To protect against this possibility, cross-validation should be used any time a variable selection routine is employed. For each data point in group B which is not involved in the regression, get the predicted Y value using the equation, and then test the correlation between the observed Y and the predicted Y at the 5% significance level. Also the MSEB and MSEA are compared to see if the former is significantly larger than the later. If that is the case, then the predicting equation is tailored for the sample data. If the correlation test is determined to be non-significant or the MSEB is significantly larger than MSEA, then the cross validation fails. If the cross validation is unsuccessful, the model has been tailored too precisely to the sample, another try will be attempted with fewer predictors.

Step 5: Best prediction equation

If the cross-validation is successful, the terminal model is run again using all the data (group A and group B) to obtain the most stable estimates of the regression coefficients.

Development of the methodology

The traditional software development models, like the waterfall life cycle model [17] and spiral model [18], provide systematic methods to separate the development process into different stages with an explicit communication boundary between two consequent stages. Most of the verification and validation of in the existing models are 1) to check if the documents are complete; 2) to see if a certain discipline is enforced; 3) to check if the documents produced are consistent with those accepted at the beginning of the phase (produced by the previous phase). This methodology provides 1) the quantitative evaluation of the design 2) feedback information to help the redesign effort; 3) improvement in software development efficiency; 4) reduction of the budget. The methodology suggested in this research incorporates all of those which are not provided by the existing models. The current ongoing research project is sponsored by Software Productivity Solutions (SPS). The software development environment used is SPS follows the spiral model [18]which includes multiple builds. The design method used in SPS is the Ada box structure design method pioneered by Harlan D. Mills [19]. The method is modified to support the object-oriented paradigm. Classic-Ada [20] is used as the programming design language in the environment. Classic-Ada is a superset of Ada. The Classic-Ada processor could parse all the Ada syntax plus nine new object-oriented constructs which supports the object oriented design concepts such as inheritance and dynamic binding. The design process is a stepwise refinement evolution with the final version of the design being the code in Classic-Ada. The Classic-Ada code is then processed by the Classic-Ada Processor developed by SPS. The MIL-STD-1815A standard Ada code is then generated by the processor. The error history data collected contains the information about the type of the error, its severity, where (which procedure or package) it was discovered, who discovered it, and who fixed it. For the software development environment in SPS, the metrics could be applied in each phase of the design, and multiple linear regression could be performed at each step of the design to reveal the relationship of different types of metrics and the maintainability in terms of number of errors.

V. Conclusion

By applying metrics to the design, the methodology which predicts the maintainability of the software product may be developed and tailored for a specific environment. There are three necessary conditions for the development of the methodology: 1) the software development process in the environment is divided into separate phases; 2) a PDL is used as the design tool; and 3) the maintenance history data of the end product is collected. With the prediction of maintainability of the software in the design phase, potential problem areas may be detected and corrected early in the software development process. Due to the early control and consideration of the maintainability, correcting problems within the design without implementing it is possible. This would reduce the development and maintenance cost and produce higher quality software product. The methodology could make a large contribution to the software development community towards their goal of achieving a quality software product which has a low development cost.

Bibliography

1. DeMarco, Tom, "*Controlling Software Projects: Management, Measurement & Estimation*," Englewood Cliffs, NJ: Yourdon Press, 1982.

2. Mills, Harlan D., and Peter B. Dyson, "Using Metrics To Quantify Development," *IEEE Software*, March 1990, pp.15-16.

3. Conte S. D., H. E. Dunsmore, and V. Y. Shen, "*Software Engineering Metrics and Models*," Benjamin/Cummings Publishing Company, Inc., 1986.

4. Halstead, Maurice H., "*Elements of Software Science*," New York: Elsevier North-Holland, Inc., 1977.

5. McCabe, Thomas J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, December 1976, pp. 308-320.

6. Henry, Sallie, Dennis Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, September 1981, pp. 510-518.

7. McClure, Carma L., "A Model for Program Complexity Analysis," *Proceedings: 3rd International Conference on Software Engineering*, May 1978, pp. 149-157.

8. Woodfield, S. N., "Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors," Ph.D Dissertation, Computer Science Department, Purdue University, December 1980.

9. Henry, Sallie, Dennis Kafura and Kathy Harris, "On the Relationships Among Three Software Metrics," *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981, pp. 81-88.

10. Henry, Sallie, and Calvin Selig, "Predicting Source-Code Complexity at the Design Stage," *IEEE Software*, March, 1990, pp.36-44.

11. Rombach, Dieter, "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990.

12. Henry, Sallie, "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers; or, Do you Recognize This Well-Known Algorithm?", *The Journal of Systems and Software*, Vol. 8 No. 1, January 1988, pp. 3-11.

13. Chappell, Bryan, Sallie Henry, and Kevin Mayo, "Measurement of Ada Throughout the Software Development Life-Cycle," *Proceedings of Eighth Annual National Conference on Ada Technology*, March, 1990, pp.525-532.

14. McCabe, T., Butler, C., "Design Complexity Measurement and Testing," *Communication of the ACM*, December, 1989, pp. 1415-1424.

15. Wake, Steve, and Sallie Henry, "A Model Based on Software Quality Factors Which Predicts Maintainability, " Proceedings: Conference on Software Maintenance-1988, pp. 382-387.

16. Kafura, Dennis and Sallie Henry, "Software Quality Metrics Based on Interconnectivity," *The Journal of Systems and Software*, Vol. 2 No. 2, June 1982, pp. 121-131.

17. Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of the 1970 WESCON*(Los Anageles, CA., Aug. 25-28), Western Periodicals Co., North Hollywood, CA., pp.A/1/1-9.

18. Boehm, B. W., "A Spiral Model of Software Development and Enhancement," *ACM Software Engineering Notes*, Vol.11, No.4, pp.14-24, reprinted in *Computer* Vol.21, No.5, 1988, pp.61-72.

19. Mills, Harlan, Richard C. Linger, and Alan R. Hevner, "Principles of Information Systems Analysis and Design," Academic Press Inc., 1986.

20. Software Productivity Solutions. Inc., "Classic-Ada User's Manual," Software Productivity Solutions Inc., Melbourne, Florida 32936.

Wei Li
Computer Science Department
Virginia Tech
(703) 231-5853

Mr. Wei Li received his B.S. and M.S. degrees in Computer Science from Peking University. He is currently a Ph.D candidate in the Computer Science Department at Virginia Tech. His research interests include software engineering, software metrics, object-oriented software system development and measurement, and large-scale software system quality control. For more information, contact: li@vtodie.cs.vt.edu.

Dr. Sallie Henry
Department of Computer Science
Virginia Tech
703/231-7584

Sallie Henry received her B.S. from the University of Wisconsin-LaCrosse in Mathematics. She received her M.S. and Ph.D. in Computer Science from Iowa State University. While attending Iowa State University, Dr. Henry's research interests were in Programming Languages, Operating Systems, and Software Engineering. After completing her Ph.D., she returned to the University of Wisconsin-LaCrosse as an assistant professor and later an associate professor of Computer Science.

Dr. Sallie Henry is currently an associate professor of the Computer Science Department at Virginia Tech. She has been working in the area of Software Engineering for the past eight years. Her primary research interests are in Software Quality Metrics, Evaluation of Methodologies, Software Testing Methodologies, and Cost Modeling. Her most recent work has been in the validation of software quality metrics, with particular focus on applying metrics during the design phase of the software life cycle. The first of two design studies uses an Ada-like PDL for designing software and the other study incorporates quality metrics with a graphical design language. Dr. Henry's research has been supported by funding from NSF, Naval Surface Surface Weapons Center, Digital Equipment Corporation, IBM, Software Productivity Consortium, Virginia's Center for Inovative Technology, and Xerox Corporation. She is a member of IEEE and ACM.

A SOFTWARE METRICS DATABASE: SUPPORT FOR ANALYSIS AND DECISION-MAKING

Shari Lawrence Pfleeger and Joseph C. Fitzgerald, Jr.

Contel Technology Center
Chantilly, Virginia

Abstract: We describe a software metrics database that is generated from a software metrics toolkit. The database is implemented as a collection of integrated spreadsheets and associated macros, so that the analyst can examine the metrics data for a particular point in time or over time, as she or he chooses. The advantages of such an approach are ease of use, flexibility, and ease of extension to other tools and techniques.

Introduction

Software metrics allow managers of development projects to monitor and control software quality. Quantitative quality goals can be established at a project's inception. Then, measurements made during development can track progress toward achieving the goals, as well as guide further development and testing. Even during maintenance, metrics can reflect the effects of changes in size, complexity and maintainability. However, both management and software engineers often balk at the additional time and labor needed to support metrics data collection and analysis. For example, the Software Engineering Laboratory at the University of Maryland reports that data collection and analysis add 7 to 8% to the cost of a project,[1] and DeMarco estimates that development costs increase between 5 and 10%.[2] Thus, tools and techniques must be available to software engineers to minimize the degree to which they are distracted by their metrics duties. Only then will metrics become a welcome part of software development.

At the Contel Technology Center, we are addressing this need in several ways. First, we recommend that the collection of metrics be tied to the maturity of the development process.[3,4] The metrics are used not only to monitor the activities in the process but also to help improve the process itself. Our recommended set of metrics is partitioned into levels, each of which corresponds to a Software Engineering Institute process maturity level. Each level of metrics allows management more insight into and control over the process and its constituent products.

Second, we provide each development project with a software metrics toolkit and underlying project database tailored to individual project needs. The toolkit contains metrics tools to collect and analyze data appropriate for the project's process maturity, development environment, and management needs and preferences. The database supports two activities. First, its contents can be used by the tools and by project managers to monitor and make decisions about the development process. Second, the database contents can be transferred to the Contel corporate historical database. The contents of the corporate database are analyzed to examine trends, make predictions, and set standards for future projects.

This paper describes both the toolkit and the project and corporate databases. It explains how commercial metrics tools are evaluated using criteria based on Contel's extensive CASE tool evaluations. The evaluation results are stored in a repository, and information is retrieved using a faceted classification scheme. Once the desired tools are selected, they are fashioned into a toolkit with a supporting project metrics database. The Contel corporate database is an amalgamation of the project historical databases. Over time, the corporate database will act as an important historical record of the way Contel develops its software, and decisions about new projects can be made based on past history, not just on expert judgment.

Tools Evaluation

To provide the metrics for a database, commercial metrics tools were evaluated using criteria based on Contel's CASE tool evaluations.[5] Two stages were involved. First, a paper evaluation addressed the characteristics of the tool described in the vendor literature and tool documentation. Second, a hands-on trial of each tool was performed, where possible. The criteria for evaluation and a list of current results are reported in a Contel technical report.[6] The evaluation results are also stored in a repository on an IBM PC, and information is retrieved using Prieto-Diaz's faceted classification scheme.[7] The faceted structure allows a project manager to select descriptions of existing tools based on descriptors such as desired metric, environment and method. For example, a manager can request all tools in a VAX environment that measure complexity using cyclomatic complexity.

By having the evaluation results on-line, the repository can always report the most current information on tools available. As tools are used in the corporation, likes and dislikes can be reported and recorded in the appropriate evaluation file. In this way, project managers can choose metrics tools most appropriate for their needs.

The Project Toolkit

Once the desired tools are selected, they are fashioned into a toolkit with a supporting project metrics database.[8] In most cases, the toolkit is built on an IBM PC, and the database is composed of Lotus 1-2-3 spreadsheets. (Where the project manager prefers other tools or a different platform, the manager is responsible for reformatting the metrics data to a form that is consistent with the corporate database.) The tools selected for the kit include a cost estimation tool called Before You Leap (BYL, from Gordon Group, San Jose, California) and a code analysis tool called PC-Metric (SET Laboratories, Portland, Oregon). The tools were chosen based on four needs:

- ease of use
- access to multiple metrics in one tool
- low cost
- availability for multiple development languages and methods

In addition, Lotus 1-2-3 version 3.0 is included, as are programs that allow users of BYL and PC-Metric to transfer both input and output data to specially-designed 1-2-3 spreadsheets. Macros are included that allow the user of the toolkit to display and analyze a variety of metrics graphically, both in snapshot mode and longitudinally for trend analysis.

The Project Metrics Database

The project database itself is a collection of spreadsheets. Version 3.0 of 1-2-3 allows multiple spreadsheets to be stored in a single file and analyzed together; the design of the database reflects this added flexibility. As each tool generates a picture of the project at a point in time, the new data are converted to a spreadsheet stored with similar spreadsheets from earlier in the development or maintenance process.

Thus, the spreadsheets are designed as shown in Figure 1, so that a set of "snapshots" of the product and process can be taken over a period of time. That is, the same metrics are measured at different points during the development process, so that trends and patterns can be observed. For example, size (in lines of code) can be traced from one spreadsheet to another over time, and the overall growth of the code can be graphed and analyzed. Similarly, the number of errors found in requirements, design, code and test can be tracked longitudinally. This analysis can lead not only to assessments of the quality of the products but also to understanding of the effectiveness of the process: where are we finding most of errors, and why?

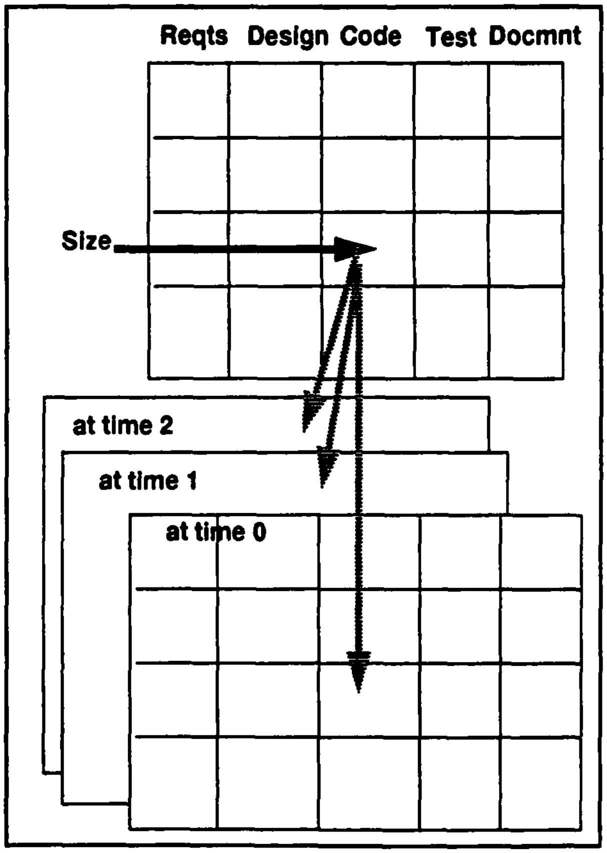


Figure 1: Multiple Spreadsheet Analysis

Project Management Spreadsheet

The first spreadsheet in the database reflects project management data and corresponds to the information required at level two of the process maturity hierarchy. Because project managers often have their own preferred project management packages, this information is entered manually by the user of the toolkit. Data requested include:

- Actual software size
- Effort to date (staff-months)
- Budget allocated
- Budget spent
- Elapsed time (days)

New metrics recorded over time are tracked within one spreadsheet, across the rows. The final column should represent the completed project.

Process and Product Characteristics

The second set of spreadsheets contains data from the BYL package. BYL uses a function points technique to estimate size, and then implements COCOMO to estimate effort and schedule. The value of BYL (and tools like it) to a project is not merely for estimation but for capturing descriptions of the requirements, the process, the product, and the personnel. Thus, we use the input to the function points calculations as a way of describing the requirements for the system. These inputs include:

- External inputs and inquiries
- External output
- Logical internal files
- External interface files

to yield an intermediate measure of total unadjusted function points

The constraints on the product are captured in the ratings for a variety of adjustment factors:

- Processing complexities
- Data communications
- Distributed functions
- Performance
- Heavily used configuration
- Transaction rate
- On-line data entry
- End user efficiency
- On-line update
- Complex processing
- Reusability
- Installation ease
- Operational ease
- Multiple sites
- Facilitates change

From these constraints is derived an adjustment factor and a calculation of total adjusted function points. Analysis of these data over time tells us how the requirements change during development or maintenance.

Next, we capture the COCOMO input parameters:

- Mode: organic, semi-detached, embedded
- Estimate of size, including amount of code that is new, modified and/or reused
- Product attributes: required software reliability, database size, product complexity
- Project attributes: use of modern programming practices, use of software tools, required development schedule
- Personnel attributes: analyst capability, applications experience, programmer capability, virtual machine experience, programming language experience
- Computer attributes: execution time constraint, machine storage constraint, virtual machine volatility, computer turnaround time

This spreadsheet tells us more about the process attributes, as well as the personnel and the development environment. As such, it satisfies many of the metrics desired for a project at maturity level 3.

The output from BYL is also stored as a spreadsheet, so that effort is estimated for each of the following activities:

- Preliminary planning
- Design
- Programming
- Integration and testing

- Maintenance

An estimated cost for the total project is also calculated and stored. In fact, the total cost is allocated to each of the following types of participants in a typical project.

- Requirements analysts
- Product designers
- Programmers
- Test planners
- Value engineers
- Project office
- Configuration management
- Writers and illustrators

From this information, estimates are made and captured for

- Software development costs
- Lifetime maintenance costs
- Total life cycle costs

Analysis of the BYL spreadsheet information can reveal much about the way in which changes in requirements or project characteristics are reflected as changes in schedule and effort. Regardless of the project manager's confidence in the ability of COCOMO to predict these variables accurately, the inputs to BYL, viewed over time, paint a picture of the volatility of certain aspects of software development and maintenance. The historical record so obtained can be used to predict behavior later in the project, as well as behavior on subsequent, similar projects.

Code Analysis Measurements

PC-Metric data constitute an additional set of spreadsheets. Here, the code is analyzed for several characteristics reflecting size, complexity, and vocabulary. In particular, for each module we capture:

- Halstead metrics n_1, n_2, N_1, N_2, N, $\hat{N}$, P/R, V, E
- McCabe complexities *VG(1)* and *VG(2)*
- Number of lines of code
- Number of semi-colons
- Number of statements

These measures are viewed both in specific instances and over time. Those modules with size or complexity considerably greater than the mean are examined to determine why; they are candidates for redesign or rewrite. Over time, the measures are compared with error data, and some metrics are used as predictors of likely candidates for more thorough testing.

Error Analysis Spreadsheets

The final set of spreadsheets captures data about errors found during reviews, walkthroughs, and testing. These spreadsheets contain the history of error discovery and correction. They can be used to calculate error density and to help determine test priorities. In addition, the error discovery information reveals a great deal about the effectiveness of the process. For example, a mature, effective process will identify many errors during requirements and design reviews, while an immature process does not uncover most of its errors until testing and integration.

The Corporate Metrics Database

The Contel corporate database is an amalgamation of the project historical databases. Added to the project information are descriptors of the development process itself, so that over the long term, we can determine which process characteristics are necessary to make a project successful. The corporate database resides on a Sun, using Lotus 1-2-3 supplemented with analysis using the Oracle database management system. This additional database capability allows us to answer questions not easily addressed with 1-2-3. For example, we can ask for all modules whose complexity exceeds a certain level, or for all projects whose error density exceeds the mean. In this way, we can analyze corporate data to determine which process activities are the most effective, how requirements volatility affects the final product, and more.

In addition to the database analysis provided by Oracle and by 1-2-3's statistics and graphics capabilities, we also perform classification tree analysis using a tool called CART (California Statistical Software, Inc.). This technique, suggested by Porter and Selby, allows managers to assess which metrics are the best predictors of characteristics such as cost or error-proneness.[9,10]

The output of a classification tree analysis is a decision tree that points out those metrics that have been most effective in the past at indicating the presence of a problem. An example classification tree is shown in Figure 2. Here, we are trying to predict which modules will have the most errors. The tree tells us to look first at size. If the size is between 100 and 300 lines of code, then we must look next at the module's complexity. If the complexity is 15 or more, the module is likely to have errors. Similarly, if a module is over 300 lines of code and has not had a design review, then if it has been changed five or more times, it is likely to have errors. The decision tree is based on a statistical analysis of data from past projects. This type of analysis will help to tell us which process activities are the most effective, which tools work best with which processes or projects, and which modules require the most concentrated testing.

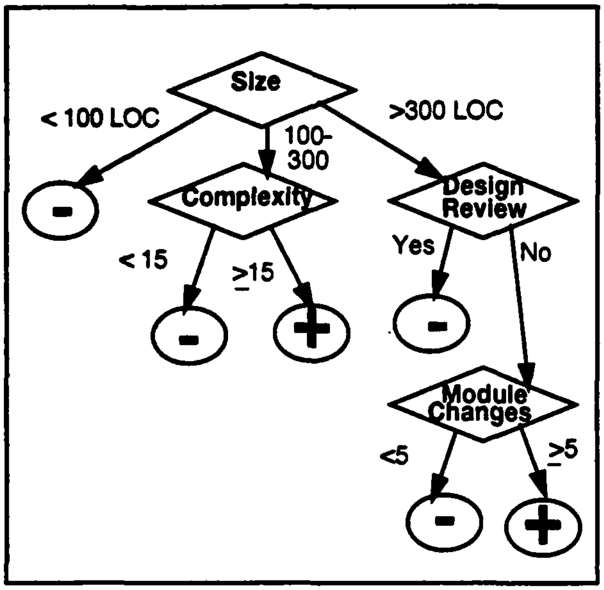


Figure 2: Example Classification Tree

Over time, the corporate database will act as an important historical record of the way Contel develops its software. Eventually, the corporate database will be used in concert with project management packages to produce more accurate estimates of effort and schedule. Decisions about new projects can be made based on past history, not just on expert judgment.

Summary and Conclusions

The metrics toolkit and databases described here are the first steps in Contel's implementation of a corporate-wide metrics program. We plan to add tools to analyze design complexity, based on work by Wayne and Dolores Zage at Ball State University.[11] In general, we hope to implement more metrics at the design and requirements stages of development, in the hope of being able to predict and fix errors early in development. Later implementations of the toolkit will contain tools to help guide testing and to predict software reliability.

The metrics toolkits and databases remove the burden of collection and analysis from the project's software engineers. Wherever possible, metrics are collected and analyzed automatically, so that metrics responsibilities are as transparent to the developers as possible. Managers can concentrate on the results of the analysis, not on the generation of information from the data collected. At the same time, the maintenance of project and corporate metrics information is relatively painless.

It is important to note that the metrics chosen are those that help to analyze the product and the process, not the people themselves. Experience has shown us that when the focus of the metrics program is process improvement, rather than on criticism of the developers themselves, people are willing (and often eager) to supply data and analyze the metrics.

The field of software metrics is relatively immature, and it is easy to refuse to collect metrics until a comprehensive and definitive set of metrics has been developed. However, such an attitude ignores the power of existing metrics to predict and guide *when the metrics program is tailored to the needs of the organization using them.* Contel's metrics databases are a giant step forward in the search for understanding and controlling software development. We anticipate that we will be adding new metrics, dropping old ones, and revising our thinking as we learn more about how our company develops software. A great advantage of the classification tree approach is that we need not collect the same metrics on all projects in order to be able to analyze trends and make decisions. Thus, no matter where we start, the project and process histories we build will always be useful to decision makers. Our corporate database lays the groundwork for future software development and maintenance at Contel.

References

1. David Card and Robert Glass, *Measuring Software Design Quality*, Addison-Wesley, 1990.

2. Tom DeMarco, Comments made at the 12th International Conference on Software Engineering, Nice, France, March 1990.

3. Shari Lawrence Pfleeger, *Recommendations for an Initial Set of Software Metrics*, CTC-TR-89-017, Contel Technology Center, 1989.

4. Shari Lawrence Pfleeger and Clement L. McGowan, "Software Metrics in a Process Maturity Framework", *Journal of Systems and Software*, July 1990.

5. Shawn Bohner, *Computer Aided Software Engineering Tools Evaluation Criteria*, CTC-TR-89-008, Contel Technology Center, 1989.

6. Shari Lawrence Pfleeger and Joseph Fitzgerald, Jr., "Software Metrics Tools Evaluation", *Contel Technology Center Technical Note* CTC-TN-090-017, September 1990.

7. Ruben Prieto-Diaz and Peter Freeman, "Classifying Software for Reusability", *IEEE Software*, 1987.

8. Shari Lawrence Pfleeger and Joseph C. Fitzgerald Jr., "A Software Metrics Toolkit: Support for Selection, Collection and Analysis", *Proceedings of the Eighth Annual Pacific Northwest Software Quality Conference*, October 1990.

9. A. A. Porter and R. W. Selby, "Empirically Guided Software Development Using Metric-based Classification Trees", *IEEE Software* 7(2), March 1990.

10. Shari Lawrence Pfleeger, Joseph Fitzgerald Jr. and Adam Porter, "The Contel Software Metrics Program", *Proceedings of American Society for Quality Control First Annual Conference on Software Metrics*, November 1990.

11. Wayne Zage and Dolores Zage, *Design Metrics*, SERC Technical Report, Purdue University, 1990.

Biographies

Shari Lawrence Pfleeger is head of the Software Metrics Project at the Contel Technology Center's Software Engineering Laboratory. She consults nationwide on software engineering, computer security, and other aspects of software systems development. Pfleeger's current research interests involve process modeling, maintenance metrics, and cost estimation. She is the author of research papers in mathematics and computer science and of two university textbooks: *Introduction to Discrete Structures* (Wiley, 1985) and *Software Engineering: The Production of Quality Software* (Macmillan, second edition 1991). She received a B.A. in mathematics from Harpur College, an M.A. in mathematics and an M.S. in planning from The Pennsylvania State University, and a Ph.D. in information technology from George Mason University. Dr. Pfleeger is a member of the ACM, IEEE Computer Society, and Computer Professionals for Social Responsibility.

Joseph C. Fitzgerald, Jr. is a member of the Software Metrics Project at Contel Technology Center, where he does research and development on software metrics. His current assignment includes evaluation of software metrics tools and the construction of a metrics toolkit for distribution throughout the corporation. On previous projects, he developed two software component library/ retrieval systems to support reuse of existing software. He prototyped other software library systems using retrieval techniques from artificial intelligence. Fitzgerald's research has included investigation of domain analysis methodologies, and the application of research results to the domain of command and control systems. He has over five years of industry experience in computer programming and software engineering. Fitzgerald received his B.S. in Computer Science from Drexel University in 1987 and is currently pursuing a graduate degree in software engineering at George Mason University. Fitzgerald is a member of the IEEE Computer Society, ACM and ACM SIGAda.

A Distributed Compilation Environment - Lessons Learned

Dr. Donald Gotterbarn
East Tennessee State University
Johnson City, Tennessee 37614-0002

Summary:
This paper is a report of some lessons learned about Ada and Ada compilation while doing research on metrics for a distributed Ada compilation system. Among the items discussed are: standard test suites for Ada programs, exceptional data structures that slow compilation time, the validity of the line of code metric, and the lines per minute standard for compilation efficiency. Some results have significance for the way we write Ada code and for the types of software engineering metrics that are appropriate for Ada.

BACKGROUND

Many Ada projects underway today require 1,000,000 to 5,000,000 lines of code. Programs of such size and complexity consume the limited resource of the CPU during compilation. The compilation speeds for Ada compilers for a 1750 processor have varied from 859 to 40 LPM (lines per minute).[1] This means that in the best case (859 LPM) 1,000,000 lines of code would take 19.4 hours of CPU time and in the worst case it would take 416 hours (17.3 days) of CPU time. The impact of recompilation in an Ada development environment can be very significant. Though Ada does allow the division of a program into small compilation units, massive recompilation can result from even a small change to a single module on which higher level packages depend.

The ways to reduce this compilation time range from buying only the fastest hardware to using smart compilation techniques.[2] To reduce the total compilation time of large (more than 1,000,000 lines of code) embedded Ada programs, a distributed parallel Ada compilation system on a network of computers was developed. Tests on this system using small to medium size (1,000 to 50,000 lines of code) Ada programs have shown various levels of performance improvement over sequential compiles.

This system reduces elapsed compilation time by performing parallel compilations of different compilation units and by distributing the compilation across processors. The system automatically distributes the compilation workload. The system first builds an order of compilation file that is used to create a module dependency graph to schedule the order of compilation. All compilation units with no compilation dependencies are distributed to available network nodes and compiled. Only components with all their dependencies compiled can be compiled in parallel. The compilation works its way up a dependency tree like the one in Figure 1 as leaf components are compiled. Component F 'withs' components A and B, so F cannot be compiled until A and B are compiled.

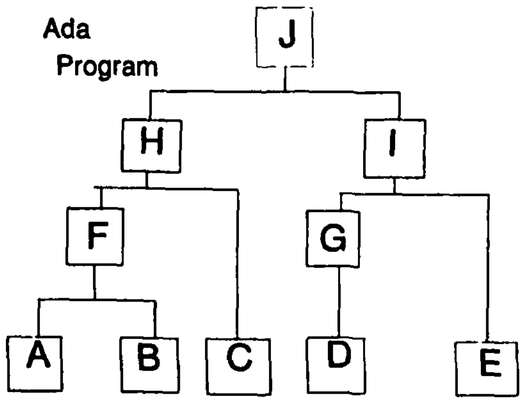


Figure 1 Dependency Tree

When a compilation unit has all the segments it is dependent on compiled then it becomes a candidate for compilation. Four Low Level Computer Software Components (LLCSC's) manage this process. The four LLCSC's serve the following functions: the interface builds a dependency graph and determines when the compilation graph has been completed, the router (R) following a scheduling algorithm decides which processor is assigned to do each compile, the server (FS) manages the workload on the nodes and the compile manager (CM) builds and manages detached processes to do Ada compilations. The choice of which network nodes to use for the distributed compilation and where the LLCSC's reside can be determined by the user before starting the compilation.

Figure 2 represents a network with six nodes on an ethemet and the LLCSC's on each node.

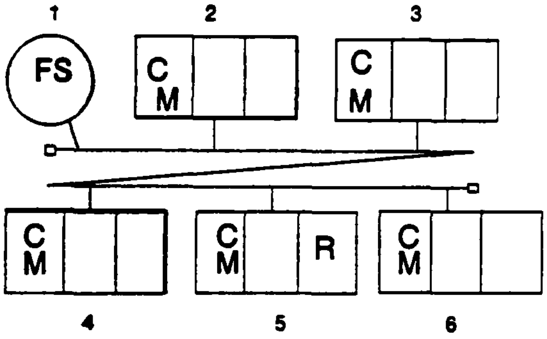


Figure 2 Distributed Compilation System

When the program in figure 1 starts its compilation on this system, the files with no dependencies are distributed to the available compilation slots as shown in Figure 3.

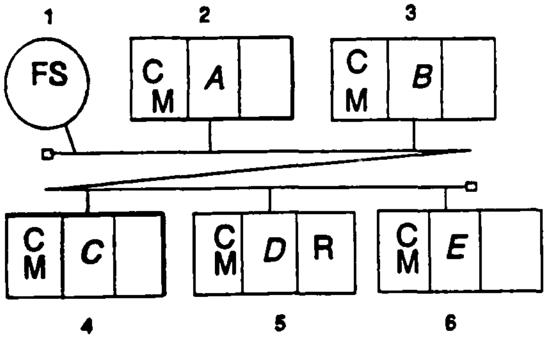


Figure 3 Initial Configuration

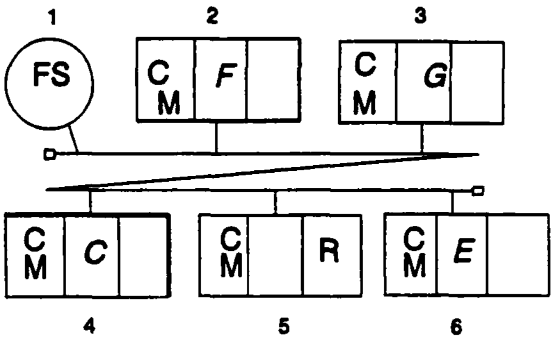


Figure 4 Units A,B,& D Are Compiled

Figure 4 shows the state of the system, when files A, B, and D have finished compiling, and files F and G were started in their place. Node 5 remains unused while waiting for the availability of a compilation unit that has all its dependencies successfully compiled.

This illustration of how the system works also shows one of its weaknesses, namely, depending on the order of compilation and the time it takes for a unit to compile, it is possible for one or more processors, like G, to be idle. The dependency relations between the Ada compilation units has a direct effect on the efficiency of the compilation in a distributed environment.

Research was done: to develop tools to predict which Ada programs would take a long time to compile on this system and to develop metrics to predict the compilation time of large Ada programs. The results would be used to improve decisions about distributed compilation configurations for different Ada programs. They also would be used to determine the feasibility of doing multiple parallel compiles on the system.
This paper is a report on some lessons learned about Ada and Ada compilation as a result of this research and not a report on the direct results of this research project.

General Approach

The initial plan had two parts. First, we wanted to find those factors which impact the compilation time of individual Ada programs compiled this system and we wanted develop a method that could predict the compilation time of a particular program before its compilation on the system. This required the development of metrics for the compilation time that are based on a static analysis of the source code. Metrics for the overhead time of the system configuration on which the program will be compiled were needed. The second part was to extend to multiple distributed simultaneous compilations on a single hardware configuration the metrics discovered for the distributed compilation of a single Ada program.

The results of these steps would be implemented in a program that scans the source code to derive the metrics and models the parallel compilation process.

Individual Package Compilation

The first part of the research plan--develop measures for compilation time predictions--was founded on several presumptions. These included: there are elements of the Ada language that can be timed; these elements can be processed in a way that simulates the distributed complexity of the system and the complexity of the modeling process was manageable.

The first step was to examine different elements of the Ada language and study their compilation times. We carefully noted features of the language that generated anomalous compilation times. A group of standard Ada test programs was used in this part of the research. ACEC programs were used and PIWG Z-tests Z000110-Z000315 were used. We gathered the timing for these programs and related them to several standard metrics -- lines of code and language structures. Because most of these tests were done in an operational rather than a controlled environment all timings were run at least four times. The timings used in the calculations were based on an average of these runs. There was less than a 10% error.

Results

Test Suites

The PIWG tests are limited because they primarily focus upon execution benchmarks rather than compilation. The Z-tests (Z000110-Z00315) include: integer arrays of 1000 elements and generic declaration. The PIWG tests did not cover many of the cases we wanted to test, such as enumeration types, float arrays, and case statements, so we extended the Z tests in two directions (Samples of these additions are in the appendix). Many of the standard test suites do not include adequate tests for compilation time. The emphasis of these test suites is on execution, for example, 'For loop' testing procedures only vary the number of iterations of the loop, which has almost no effect on compilation time. The failure to include the compiler directive Pragma, is a serious deficiency in these tests.

Linear Growth with Size

Most of our timing tests produced a smooth linear growth in compilation time as the metric increased, indicating that any one of the metrics could be used to give a reasonable indication of compile time. The timing results of using lines of code or data structures were predictable. There were, however, some significant anomalies. When dealing with initialization of integers, floats and enumerated variables, the increase in compilation time (represented vertically) was directly proportional to the increase in the number of elements. The slopes of the types tested were similar. Figure 5 shows the results for integers, floats and enumerations.

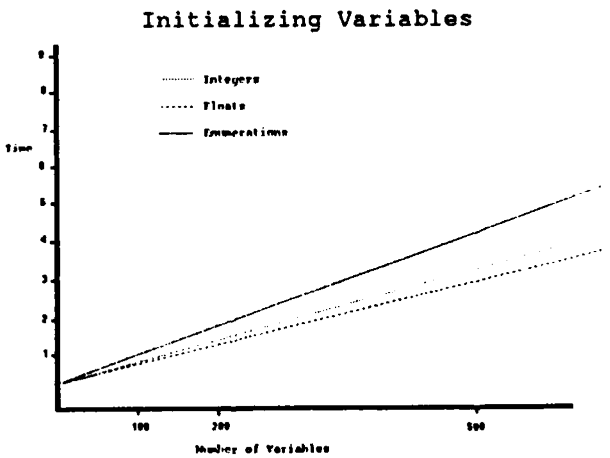


Figure 5 Initializing Variables

When testing array initializations the slopes were consistent as the numbers of variables increased, but the mode of initialization of the array significantly changed the angle of the slopes. The compilation time for arrays that either did not initialize their elements or initialized all positions in the declaration, increased gradually as the number of elements increased. Initialization of elements by position in the declaration also increased slightly in compilation time as the number of elements increased. The compile time for initialization by name, however, in either the body or the declaration increases significantly as the number of array elements increase.

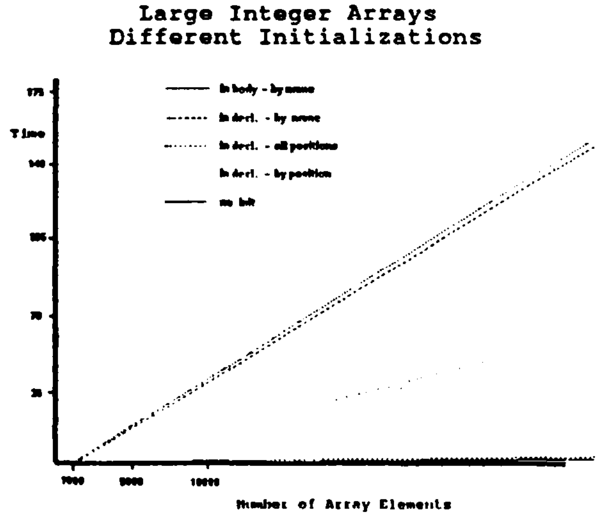


Figure 6 Array Initializations

The timings for case statements and for loop statements produced linear growth. The only problem encountered for case statements was that the VAX Ada compiler no longer optimizes case statements that exceed 100 cases. Surprisingly, overloaded procedures produce a linear growth out to about 250 overloaded procedures. After 250 procedures the growth is exponential and then it is vertical at about 1,000 overloaded procedures.

In most cases the compilation time for different elements in the language increased as the number of elements being compiled increased. The rate difference in the rate of increase between the elements points out the inequality of operators and operands from the point of view of the compiler. This result raises serious questions about any metric, like Software Science that treats operators and operands equally.

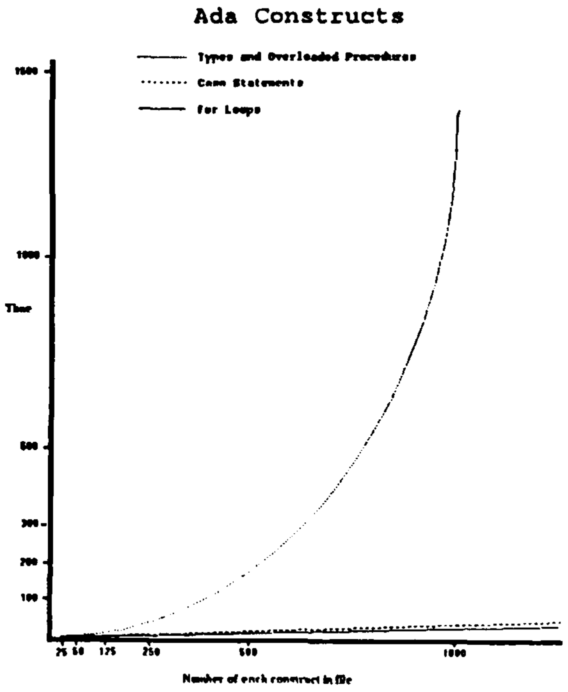


Figure 7 Ada Constructs

Operator Counting

Although operator counts are used as a metric for the difficulty or amount of time needed to write a program, the same metric cannot be easily used as an indication of compile time. We can see from the sample in figures 5-7 that there is a significant difference in the compilation times of different operators. It would be a mistake to treat them all as if they had a similar weight. The use of operator counting in Ada is very complex. Each operator must be assigned a different weight according to compilation time; but first we must decide what to count as an Ada operator. Ada's unique capabilities such as overloading operators and generic instantiations makes it difficult to apply operator oriented metrics. Not only is there a difference between operators, but our tests showed a significant difference in the compilation times for different types. Instantiations of a generic integer were quicker than instantiations of a generic floating point types. The range and accuracy of the instantiated types also had a direct effect on compile time.

Lines of Code

In testing lines of code as a compilation time metric, we used the standard of counting non-comment, non-embedded semicolons as lines. We found lines of code produced a relatively consistent metric. Packages with the similar numbers of code lines took approximately the same amount of time to compile. There were two interesting results here.

The exceptions to this pattern were similar to the exceptions to the patterns of the compilation time's linear growth with linear growth in the quantity of a data type's elements. That is, we had anomalous results when we looked at elements unique to the Ada language. For example, the compilation of two line generic instantiations took as long as the compilation of 900 to 1,200 LOC packages that had no generic instantiations.

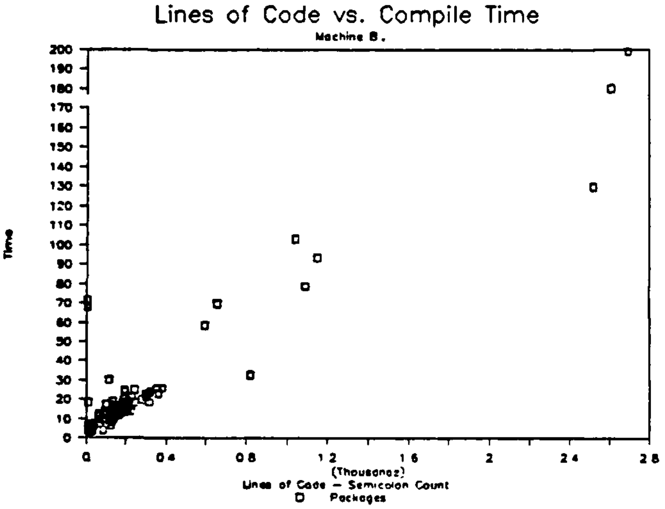


Figure 8 Lines of Code & Compile Time

We had to look at compile time on different machines. One useful result here was a proof that compilation time is directly proportional between machines. For example, all test programs compiled on a DEC 3500 took approximately 65% less time to compile than they did to compile on a MICROVAX. This has a testing advantage in that all compile test do not have to be run on all machines. The timing for different machines can be easily determined by determining the ratio of the difference between the two machines. We compiled the same set of Ada packages with 2 to 2,800 lines of code on two machines. The slopes for compile times for this sets of packages on different machines is congruent.

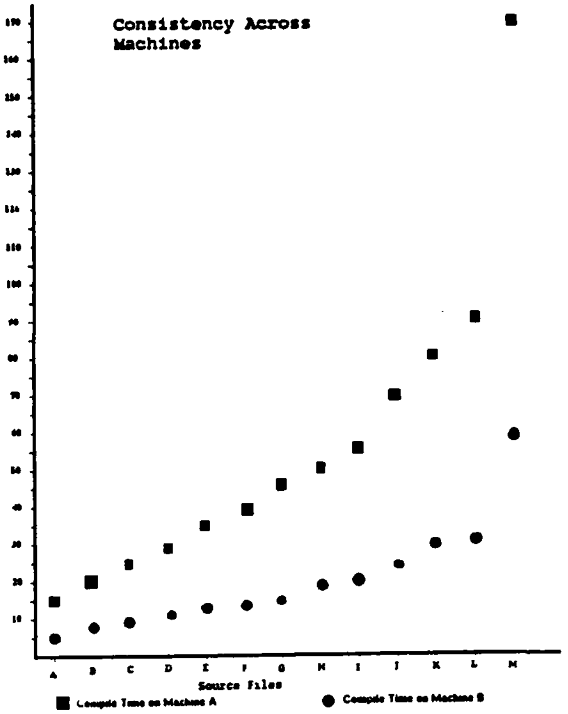


Figure 9 Consistency Across Machines

Compiler efficiency- LCPM

Many compilers give a 'Lines of Code Per Minute (LCPM)' statistic at the end of a compiler listing. Using this as a measure of compiler efficiency is misleading. The initialization of a) a real array of 50,000 elements by name in the declaration takes considerably longer than the initialization of b) a real array of 5 elements initialized by name in the declaration. The LCPM for a) is 2 while the LCPM for b) 200. The compiler is not operating less efficiently in case a), the compiler is simply doing more work per line and it is working as efficiently as in the 5 element initialization. To make the LCPM metric useful, the size of the operators and which operators are being compiled must be considered. LPM is a misleading metric because it does not reflect the number and type of elements compiled.

Compile Times are NOT additive

The next step was to derive a formula to calculate the system's overhead time in a distributed compilation. When calculations for sequential compiles were made using predicted timings for programs containing multiple packages, it was shown that the total times for compilation were less than the total of the combined times for each of the compilation sub-units in the program. This time savings was attributed to reduced paging of the Ada compiler between the compilation of sub-units. This reduction was consistent so metrics could be developed to take account of it in our calculations.

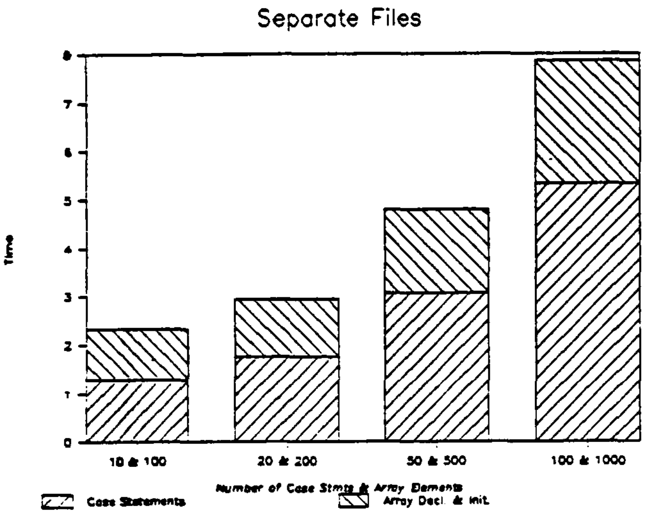


Figure 10 Separate Compilation Times

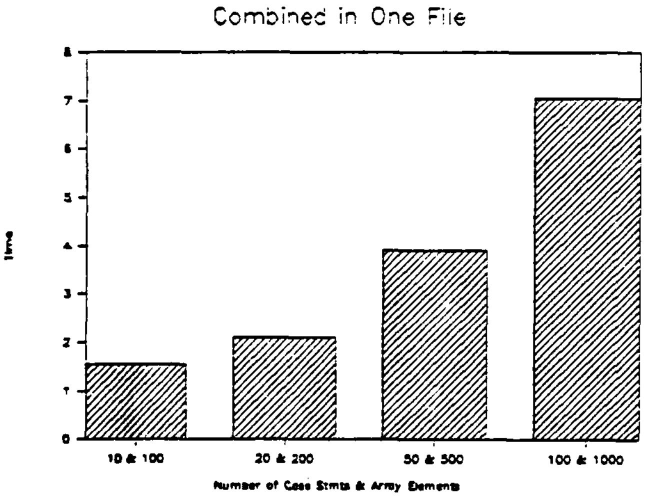


Figure 11 Combined Package Times

The times for packages compiled as separate files are shown in Figure 10. The timing results of combining these packages together into a single file before compilation is shown in Figure 11. In each case the combined file compiled in less time than the combined total time for the two separately compiled packages.

There were several bi-products of this work. It was shown that an increase in page size shortens compilation time, even for small programs. PIWG Z-test did not cover several compilation boundary conditions. There are some programming suggestions. Given the results of our analysis of array compilation timing, as the number of array elements increases, it would be better in terms of compilation time not to initialize by name. If very large arrays are initialized, it may appear that the compilation has slowed considerably because the compilation time is sometimes reported in lines compiled per minute. This metric is misleading because it does not reflect the number of elements being initialized. If two arrays were initialized with one having 1/10th the number of elements as were in the large array, then using lines per minute it would look like the second compile were more efficient. The relationship between compilation times for the same program on different machines is very important. The proportional time for compilation is consistent across different machines. Given a single set of compilation times for a large number of programs, one can easily predict a programs compilation time on another class machine. Determining the compilation times for two or three programs on the new class of machines will yield a proportional relationship to the compilation times on the other class of machines. From this ratio, the compile times on the new class of machines can be predicted.

APPENDIX

Z TESTS, PIWG PROGRAMS Z000110 - Z000315

Subject:
Description:
Size, Item counted:

Program
collection of various packages
24 pkgs., 2907 lines

Integers
all initialized to 1 in declaration
individually initialized in declaration
100, 200, 500, 1000 variables

Integer arrays
initialized by name in body
100, 200, 500, 1000 array elements

Chained access types
type_(n) is access type_(n-1);
in private part
100, 200, 500 types

Null procedures
procedure declarations and bodies
100, 200 procs

Instantiations
instantiating INTEGER_IO(INTEGER);
100, 200 new pack.

Tasks
specifications with 2 entries each
10, 20, 50, 100 tasks

Renames
chained procedure renames
100, 200, 500 procs

With, Use
withing TEXT_IO
withing & using TEXT_IO
100, 200, 500 withs
100, 200, 500 with, use

Nesting
blocks
blocks that define & raise exceptions
procedures
packages
tasks
20, 50, 100, 200 blocks
10, 20, 50, 100 blocks
10, 20, 50, 100 procs
200 packages
200 tasks

Overloading
types and overloaded procedure name
250 types and procs

Procedures
declarations only
10, 20, 50, 100 procs

Packages
package with one procedure declaration
package with one rename
10, 20, 50, 100, 200 pack.
10, 20, 50, 100 pack.

Generics
generic procedure declarations
10, 20, 50, 100, 200 procs

X TESTS, EXTENSION OF THE PIWG Z-TESTS

Subject:
Description:
Size, Item counted:

Floats
all initialized to 1 in declaration
individually initialized in declaration
no initialization
100, 200, 500, 1000 variables

Float arrays
initialized by name in body
initialized by name in declaration
initialized all positions in declaration
initialized by position in declaration
no initialization
100, 200, 500, 1000 array elements

Enumeration
3 & 12 symbols
3 & 12 symbols, 1 type
3 & 12 symbols, 1 type,
individually initialized in declaration
100, 200, 500, 1000 types
100, 200, 500, 1000 variables
100, 200, 500, 1000 variables

Case
3 options plus "others"
10, 20, 50, 100 statements

For
loop 1..10 with 1 assignment statement
10, 20, 50, 100 loops

Z-PLUS TESTS, EXTENSION OF PIWG Z-TESTS

Subject:
Description:
Size, Item counted:

Integers
no initialization
100, 200, 500, 1000 variables
5000, 10000, 50000 variables

Integers arrays
initialized by name in declaration
initialized all positions in declaration
initialized by position in declaration
no initialization
100, 200, 500, 1000 array elements
5000, 10000, 50000 array elements

Constraints

The system investigated is the Ada Distributed Compilation System running on a VAX Cluster at Boeing Military Airplane (BMA), Wichita Kansas. The operating system used for this project is DEC VMS 5.1 and the Ada compiler is VAX Ada 1.5. The hardware systems on which tests were conducted consist of two distinct clusters. One cluster, at BMA, contains VAXSTATION 3100's MicroVAX I's, and DEC 3500's. The other system, at The Wichita State University consists of several VAXSTATION 3100's and a DEC 8650.

[1] Eys, Carlene "Ada Distributed, Parallel Compilation," 1988

[2] Tichy, Walter F. "Smart Recompilation," ACM Transactions on Programming Languages and Systems, Vol 8, No. 3, 1986. p 273ff

BIOGRAPHICAL INFORMATION

Educated at the University of Rochester, Dr. Gotterbarn taught for several years at such schools as the University of Southern California and Dickinson College. He has also worked as a computer consultant. Among the software projects he was responsible for were a nationwide videotex system, several database systems for the U.S. Navy and the Saudi Arabian Navy, and an interactive crime reporting database. He is currently at East Tennessee State University where he teaches software engineering courses, data communications, and database. As a Visiting Scientist at the Software Institute, he developed educational materials for software engineering. His current research on performance prediction for a distributed Ada closure was supported by a contract with Boeing Military Airplanes.

INCREMENTAL OPERATIONAL SPECIFICATIONS FOR THE VERIFICATION OF ADA PROGRAMS

William Howden
Bruce Wieand

Department of Computer Science and Engineering
University of California, San Diego

Abstract: Our approach to verification involves the use of analyzable specifications embedded in programming language comments. The specifications allow programmers to add information to a program that is either difficult or impossible to extract from the program itself. Experimenting with a heavily tested operational flight program written in assembly language, we established the viability of our method by detecting several errors. We are currently extending our method to the Ada programming language. Ada introduces the potential for errors at different levels, and we have proposed extensions for handling some of these. The Ada analyzer is currently under development and will be tested on a large production Ada program. As we experiment with the Ada analyzer, we plan to add capabilities including timing analysis, the automatic generation of specifications, and partial code generation from specifications.

1. Introduction

The software development process can be viewed as one of translation from concept to requirements, from requirements to specifications, and from specifications to source code. Each step involves the translation of one description of an abstract object to another. Verification is typically defined as the checking for completeness and consistency among the various descriptions. We are concerned with the verification of programs and specifications.

In typical development environments, the transition from specifications to source code is special, because it is at this point that the abstract object description moves from the domain of human understandability to machine understandability. In making this transition, information is lost or obscured in the formalism of the machine's language. During the development process, this may not be a problem since a programmer can remember connections between specifications and a program. But the loss of information can be detrimental to program maintenance. Comments help the situation by allowing the programmer to retain pieces of the specification in the source code. Additionally, comments allow the programmer to document facts known and assumptions made about the state of the program during the coding process. However, because the coding process is a human process, it is subject to errors, and the pieces of information in the comments may be inconsistent.

We have developed a simple specification language which allows programmers to add specifications to the comments portion of a program. Our specifications are termed "incremental operational specifications" because they represent pieces of information that are added over a period of time and because they refer to the state of the program at various points on its execution paths. The specifications are in the form of assumptions and assertions. Assumptions document facts about the program that the programmer *assumes* to be true, and assertions document facts about the program that the programmer *knows* to be true. The assertions and assumptions are inserted in the program at the locations where the facts are known or are assumed. Verification consists of traversing the execution paths and checking the consistency of the assumptions against the assertions.

The approach we have developed is called QDA for Quick Data Analysis. Section 2 briefly overviews the QDA approach.[1] We have tailored QDA into a system called QDA2 for analyzing AYK-14 assembly language programs. Section 3 talks about our experience with QDA2.[2,3] We are currently extending our approach to Ada in a system which will be called QDAA. Section 4 addresses some of the difficulties in taking QDA from assembly to Ada. Section 5 presents the modifications we have planned for QDAA, and section 6 outlines our thoughts on future extensions.

2. The QDA Approach

The underlying concept in our approach is the association of an object with properties. Objects may represent a wide variety of things such as program variables, program locations, and abstractions. Properties are characteristics of objects which describe their state. For example, a property of a program variable could be "set", a property of a task entry could be "called", and a property of the abstract object "landing_gear" could be "down". There are three ways of stating the association of an object with its properties: is, iss, and nis. If "x" is an object, and "a" is a property, then "x is a" means that x has property a and possibly other properties, "x iss a" means that x has property a and no other properties ("is and only is"), and "x nis a" means that x does not have property a. Specifications are built from these atomic relationships using logical connectives.

Specifications are either assertions or assumptions. An assertion is a formula preceded by a "!" symbol, and an assumption is a formula preceded by a "?" symbol. The formula parts of specifications are either rules or sets of atomic relationships connected by "&" symbols (conjunction) and "|" symbols (disjunction) in disjunctive normal form. A rule consists of two formulas separated by a "->" symbol with restrictions. The restrictions are that disjunction is not allowed in the rule's left side, and its right side contains only a single atomic relationship. The following are examples of specifications.

```
! start is called
? x iss set
! x.bit1 is 1 -> landing_gear is down
```

The intent of the first specification is the assertion that a task entry "start" has the property "called", the second is the assumption that a variable "x" has the property "set" and no other properties, and the third is the rule which means that whenever a variable "x.bit1" has the property "1", the abstract object "landing_gear" has the property "down".

The above properties are all examples of what we call *dynamic* properties. Dynamic properties may change during the execution of a program. Objects may also have fixed properties which do not change. For example, the variable GWDRG2 may represent the stage 2 drag curve for a particular weapon throughout the execution of a program. This fact is asserted with the following *type* specification.

```
! type[GWDRG2] iss Drag_Crv_Stage_2
```

The word "type" is intended to represent a much more specific data type than the integer and real data types of conventional programming languages.

Assertions and assumptions are associated with the program locations at which they appear. Rules and type assertions are associated with the entire program and appear in data definition files. In addition, there are input and output specifications which may appear at the beginning of subroutines and are treated both as assertions and as assumptions. During the analysis of a particular subroutine, its input specifications are treated as assertions at the beginning of the subroutine and its output specifications as assumptions to be verified at each return point. When one subroutine is called during the analysis of another subroutine, the roles of the input and output specifications are reversed. At the point of the call, the input specifications become assumptions and the output specifications become assertions.

Verification is accomplished in three passes. The first pass creates an abstract program representing the division of the program into subroutines and the control flow within each subroutine. The second pass traverses each path in the abstract program and creates a finite state table in which each state reflects the assertions encountered along paths through the program. The third pass attempts to verify each assumption with respect to the states at the location of the assumption in the abstract program.

3. Experience with QDA2

Our first analyzer, QDA1, was an experimental analyzer which we used to develop our ideas.[4] From that came QDA2 which was briefly described in the preceding section. QDA2 was used to verify parts of the operational flight program (OFP) for the Navy's AV-8B. The OFP consists of roughly 70,000 lines of code. The majority of the code is AYK-14 assembler, and the rest is CMS-2. Approximately one third of the program contains data definitions, and the other two thirds contains about 650 subroutines. The program is well documented. In fact, the design of the QDA2 specification language is partially based on the structure of the comments. Take, for instance, the following code fragment.

```
LK  R8,GWDRG2 . PTR TO DRAG CURVE 2
              . (ARG FOR GWSDC)
JLR R4,GWSDC  . STORE DRAG CURVE 2
```

In this fragment, register R8 is loaded with the address of a weapon's stage 2 drag curve table in preparation for a call to GWSDC which stores a drag curve in it. The comment at the first instruction indicates the programmer's assumption that GWDRG2 is a pointer to the stage 2 drag curve. The QDA2 specifications for this code fragment are shown below and are enclosed in "#" symbols.

```
LK  R8,GWDRG2 . PTR TO DRAG CURVE 2
              . (ARG FOR GWSDC)
              . #? type[GWDRG2] iss
              .    Drag_Crv_Stage_2#
              . #! R8 iss GWDRG2#
JLR R4,GWSDC  .  STORE DRAG CURVE 2
```

The first comment specifies that the type of GWDRG2 is the stage 2 drag curve. The second is used to indicate that register R8 now contains the value of the variable GWDRG2.

Analysis of the AV-8B program revealed two general kinds of errors: coding errors and documentation errors. An example of a coding error concerned the use of the two-argument root-sum-square routine, MDRS2. The error was the assumption that the return value from MDRS2 was in a single register when actually it was in two registers. The relevant portions of the program with QDA2 comments is shown next.

```
JLR R4,MDRS2  . ROOT SUM SQUARE
              . RESULT IN R2
NOP           . #? R2 iss Root_Sum_Sq#
LR  R3,R2     . PLACE ROOT IN R3

...

MDRS2*        . #output:
              .  R2 iss Root_Sum_Sq_MSB &
              .  R3 iss Root_Sum_Sq_LSB#
```

At the point of the call to MDRS2, the program state was modified by adding the property "Root_Sum_Sq_MSB" to the object "R2" and the property "Root_Sum_Sq_LSB" to the object "R3". When the analyzer tried to verify the assumption at the NOP instruction, it failed. (The NOP instruction was inserted by us to emphasize that the assumption is associated with the results of the call.)

An example of a documentation error involved an anomaly between the sine and cosine of an angle. The data definition contains the following variable with QDA2 comment.

```
VRBL NHPCBS A 16 S 15 P 0
          . SIN HARP ANG BST
          . #! type[NHPCBS] iss
               Sin_Harp_Ang_Bst#
```

A usage of this variable in the program with the QDA2 comment that caught the error is shown next.

```
L  R15,NHPCBS . COS HARP ANGLE BST
              . #? type[NHPCBS] iss
                   Cos_Harp_Ang_Bst#
SU R15,DDSHRP . SIN OF HARP ANGLE
              . #? type[DDSHRP] iss
                   Sin_Harp_Ang#
```

In this case, the usage of the variable NHPCBS was correct in the program, while the original comment was wrong. Although some would say that this is not really an error, we maintain that the detection of these kinds of errors is critical for the maintenance of the program.

4. Extending QDA to Ada

We are currently extending the QDA philosophy to Ada in a system called QDAA. Parts of Ada allow easy extension of QDA2 and other parts are more difficult. Global data can still use the type specifications of QDA2. Sequential and branching statements (including if-then-else, case, and loop) can still be modeled by the finite state table approach of QDA2. And subprograms can still use the input/output specifications for their interfaces. The parts of Ada which require modification to the QDA2 approach are context clauses, block structure, parameters, overloading, task synchronization, exceptions, and generic units. These are discussed below. The following section presents some specific enhancements to QDA2 for dealing with the problems.

One of the basic problems is introduced by the free format of Ada programs. In AYK-14 assembler, the format was fixed, and associating QDA2 specifications with program instructions was easy. In Ada, comments can appear between any pair of lexical elements. This has necessitated changes in the structure of the abstract program so that QDAA specifications can be associated with parts of statements. These changes are invisible to the programmer, however, and no enhancements to the QDA2 language are required.

Context clauses allow a programmer to use abbreviated names in one module to refer to objects in another. Working under the umbrella of a context clause, the programmer is free to ignore the prefixes of expanded names. In order to allow the programmer to use names in QDAA specifications that refer to identical names in the program, we have to associate the names in the specifications with the program's expanded names so that the correct property lists are accessed. This association demands a finer degree of program modeling than QDA2 has. In QDA2, the objects in the specifications are linked to program objects only in the mind of the programmer; the analyzer does not correlate objects in the two domains. In QDAA, the analyzer will require some degree of correlation. In fact, solutions to all of the difficulties introduced by Ada involve a closer relationship between objects in the program and objects in the specifications.

The important implication of block structure is the dynamic scope of program names. This allows names defined in an inner scope to hide names in an enclosing scope. Handling this in QDAA requires changes in both the abstract program and the finite state models of the program. The abstract program has to reflect scope changes, and the process of building the finite state table has to use this information to access the correct object.

The subroutines in AYK-14 assembler pass parameters either through registers or global data. and the parameter names are identical in both the calling and called subroutines. Ada parameters require binding formal and actual parameter names so that the correct property lists are accessed. This involves two problems. (1) If a parameter is mentioned in an interface specification, its mode has to be consistent with the specification. (2) The analyzer must be able to bind formal and actual parameters using either named, positional, or default notation.

Ada's overloading feature presents the problem of name resolution. The problem is similar to the block structure problem. The abstract program must maintain the structure of subprogram declarations (number of parameters, type, order), and the finite state table building process must access this information to obtain the correct interface specifications. Overloaded operators are treated the same way.

Tasking in Ada can result in synchronization errors such as deadlock, race conditions, infinite waits, hung rendezvous, and order of elaboration, entry calls, and termination. The naive approach to some of these problems would be to analyze all possible executions of tasks. The argument for this approach would have to include limiting the size of tasks and the number synchronization points. Instead, we intend to provide techniques for specifying in one task what is assumed to be going on in another. For instance, suppose that task A waits on entry 1 before waiting on entry 2. The naive approach would specify this condition in task A and check that no execution could lead to entry 2 being called before entry 1. Instead, we assume that the programmer is familiar with the task interaction design, and we will provide facilities for documenting this knowledge so that all possible execution sequences don't have to be checked.

Exceptions cause relatively non-structured transfers of control between frames. What is required is the capability to specify the characteristics of exceptional conditions. The characteristics that must be specified are whether a frame can raise an exception and whether a frame can handle and/or propagate an exception. QDA2 interface specifications will be enhanced to do this.

Generic units allow parameterized packages. This introduces problems similar to the problems with overloading and context clauses. The solution is also similar. One key additional enhancement requires the instantiation of specifications to match the instantiation of generic units.

5. Specification Enhancements for Ada

Extensions to the QDA2 language have been proposed which solve some of the difficulties mentioned above and increase the flexibility of the specification language. Scoped specifications deal with the block structure of Ada. Inference assumptions allow logical implications to be checked. Parameter specifications deal with subprogram parameters. Specification instantiation applies to generic units, and CTL-like specifications are used for temporal aspects of the language such as task synchronization and operator sequencing. Some of the properties of these extensions are described below.

In QDA2, an assertion associated with an executable statement can only hold within the subroutine containing the executable statement. In general, this will be the case for QDAA with a few exceptions. The Ada counterpart to the subroutine is the secondary unit (i.e., the body of either a subprogram, a package, or a task). An assertion associated with an executable statement in a secondary unit can only hold within the scope of the object referenced by the assertion unless that scope extends outside of the secondary unit. In that case, the assertion can only hold within the scope of the secondary unit. An assumption in QDA2 is only checked at the point in the program where the assumption occurs. In QDAA, we have allowed assumptions to have scope so that they can be checked at every program location within a particular scope. The syntax of a scoped assumption has the keyword "$scope" in between the assumption symbol and the formula part. For instance, the assumption in the following declaration

```
available : boolean;
     --#? $scope available is set#
```

is checked at every point within the scope of the declaration. A smart analyzer may realize that this assumption only has to be checked when the properties of the object "available" change, but the semantics of scoped assumptions are that they are checked at every point within their scope.

In QDA2, a rule assumption is an assumption about the rules themselves, not the states at the point of the rule assumption. For example, the assumption

```
? x is a -> y is b
```

checks to see if there exists a rule whose left side contains "x is a" and whose right side is "y is b". (Note that rules in QDA2 are restricted to atomic specifications on the right and no disjunction on the left.) With QDAA, in addition to rule assumptions, inference assumptions will be used for checking logical implications. For instance, the inference assumption

```
? char nis available =>
                  available nis true
```

is true if and only if either the object "char" has the property "available" or the object "available" does not have the property "true". Inferencing combined with scoping provides a powerful high-level assumption mechanism. For example, the assumption

```
? $scope char nis available =>
                  available nis true
```

allows the programmer to make the assumption once at the beginning of a block instead of having to state this assumption at every point in the block where the properties of the objects "char" and "available" are changed.

In assembly language, parameters passed between subroutines are known by the same name in both the calling and called subroutines. This makes it easy to transfer properties across the interface. In Ada, actual parameters are associated with formal parameters in subprogram and entry calls. So the interface specifications for QDAA have to be able to associate formal objects with actual objects. During the analysis of a particular subprogram or entry, nothing different is required for the interface specifications. But when a subprogram or entry is called, QDAA makes connections between formal parameters and actual parameters for those formal parameters which are referenced by interface specifications. An example follows.

```
procedure P (R : in real);
     --#? $input R is positive#


...


P(S);
```

At the point where procedure P is called, QDAA would associate the actual parameter S with the formal parameter R. The assumption would check that object S has the property "positive". In the case where the actual parameter is an expression, say something like "S + T*2", QDAA would look for an abstract object named "S + T*2" with the property "positive".

An aspect of Ada that has no counterpart in assembly language is genericity. For specifications that apply to all instances of a generic unit, there is no problem. But for some generic units, the programmer may want to assume different facts about the various instances. Instantiated specifications solve this problem. If the QDAA keyword "$generic" appears in a specification in a generic unit, then, when the generic unit is instantiated, the keyword is also instantiated to the name of the generic unit. For example, suppose that there is a "read" procedure in a "messages" package with the following specification.

```
generic
    ...
    procedure read;
        --#! msg is from_$generic#
    ...
end messages;
```

Further suppose that the package is instantiated twice as follows.

```
package red_messages is new
    messages(blue_devices.display);
package blue_messages is new
    messages(red_devices.display);
```

The resulting specifications for the "read" procedure in the two instances of the package would be the following.

```
procedure read;
    --#! msg is from_red_messages#
procedure read;
    --#! msg is from_blue_messages#
```

This allows the programmer to make assumptions about specific instances of the generic unit.

QDA2 is geared towards data usage analysis and is based on simple propositional logic. One of the QDAA extensions involves operator sequence analysis. To allow reasoning which involves sequences of operators, we will base QDAA on a version of temporal logic called Computation Tree Logic (CTL).[5] The CTL temporal operators are AF, AG, EF, and EG. For example, if "f" is a QDAA formula, "AF(f)" is true at program location "p" if and only if "f" is true at some location on every path leading out of "p". The other CTL operators have similar semantics. Their common property is that they all refer to program paths *leaving* a particular program location. We have added the complementary operators, AP, AQ, EP, and EQ which refer to program paths *entering* a particular program location. With this set of temporal operators, we can concisely specify program operator sequences. This is especially useful in task interaction. For example, if it is required that the entry "put_line" in a task always completes its rendezvous, then the following specification verifies this.

```
accept put_line;
    --#? AF(put_line is returned)#
```

As another example, suppose that a task has a "start" entry which should always be called first to initialize the task before any other entries are called. The following specification placed at the accept statements for the other entries would verify this.

```
accept entryX;
    --#? AP(start is accepted)#
```

6. Future Directions

The current status of the QDAA project (as of November, 1990) is that we are finalizing the design of the QDAA specification language, and we have started work on the design of the analyzer. After this, it remains to build the analyzer, experiment with it, and upgrade it according to the experience we gain in classifying errors found in Ada programs.

The QDA2 analyzer has a hand-coded parser. We currently have a lex/yacc specification for an Ada parser that we are modifying to parse QDAA specifications. We are also modifying the abstract program structure and the finite state structure that QDA2 uses in order to handle the free format and block structure of Ada.

For experimentation, we will use a production piece of software with enough complexity to make human comprehension difficult. We currently have two possibilities. The programming team for the AV-8B is experimenting with an Ada implementation of a subset of the OFP. They have already defined the tasks involved, have selected the subset of functions to be included, and have developed the data interfaces for the packages representing modules from the CMS-2/AYK-14 implementation. The drawback is that the program is still under development. Another option is the procurement of the flight control program for a commercial aircraft.

After experimentation, we expect to make enhancements to QDAA. Some of the ones we are already considering include a more interactive user interface, timing analysis, the automatic generation of specifications, and the generation of partial code from specifications. QDA2 has a very simple user interface which simply prompts for source code files to analyze, analyzes them, and generates error reports. An enhancement currently being made to QDA2 is to allow the programmer to browse through the source code and interactively insert specifications and examine the program states. QDAA will also have this feature. Timing analysis will require the analyzer to have arithmetic capabilities. We have avoided this in the past because of theoretical complexity issues. But recently we have been working on a rudimentary arithmetic capability that can be used for simple kinds of timing analysis. Automatic generation of specifications applies to assertions. Many assertions can be derived from the code; in particular, operator events such as entry accept statements, entry and subprogram call statements, and entry and subprogram return statements. Automatic generation of assertions is desirable for two reasons. We found that the process of adding certain kinds of specifications to code can be tedious. The other reason is that humans make errors in this process. We will determine which kinds of specifications can be generated from code and how useful they are. The QDA approach to program verification has focussed on the programming and maintenance portions of software development. The automatic generation of partial code from specifications is more of a design issue. We feel that even as early as the design phase, QDA-style specifications can be written to reflect design decisions such as the presence of data and/or subprograms or the order of rendezvous points within a task. When the design phase completes, QDA design level specifications could be used to generate library unit interfaces.

References

1. W.E. Howden, *Comments Analysis and Programming Errors*, IEEE Transactions on Software Engineering, January, 1990.

2. W.E. Howden, C. Vail, D. Nesbitt, B. Wieand. *Design and Experimental Use of a Verification System Using Incremental Operational Specifications*, UCSD Department of Computer Science and Engineering, December, 1990.

3. W.E. Howden, D. Nesbitt, C. Vail, B. Wieand, *Verification of Complex Systems Using Incremental Operational Specifications*, to appear in Information Sciences, June, 1991.

4. W.E. Howden, C. Vail, R. Westbrook, B. Wieand. *User Guide and Reference Manual for the Quick Data Analyzer (QDA)*, UCSD Department of Computer Science and Engineering, Technical Report CS88-136, November, 1988.

5. E.M. Clarke, E.A. Emerson, A.P. Sistla, *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems, April, 1986.

6. Grady Booch, *Software Engineering with Ada*, Benjamin/Cummings, 1987.

7. Ken Shumate, *Understanding Concurrency in Ada*, McGraw Hill, 1988.

8. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A, January 22, 1983.

9. *AdaOFP Top Level Design Document*, Rough Draft, November 6, 1989.

William E. Howden receiveḋ the Ph.D. degree in Computer Science from the University of California at Irvine. He also holds master's degrees in both Mathematics and Computer Science and a bachelor's degree in Mathematics. Dr. Howden is a Professor of Computer Science at the University of California at San Diego. His principal area of research is program testing, an area in which has has written one book and co-edited two others. His current work is on defect analysis, a technique for detecting the occurrence of errors in large, complex systems which are only informally and incompletely specified. He has written numerous papers on program design, programming environments, and man-machine interaction. He is on the editorial board of the *IEEE Transactions on Software Engineering* and is a member of *IFIPS* Working Group 10.4, Reliable Computing and Fault Tolerance.

Bruce Wieand is currently working towards the Ph.D. degree in Computer Science at the University of California at San Diego. He holds a master's degree in Computer Science from the California State University at Fullerton, and a bachelor's degree in Computer Engineering from the University of California at San Diego. He has been working in the area of program testing in collaboration with Dr. Howden for 2 years and has co-authored several papers with him.

Both authors may be reached at the following address:

University of California at San Diego
Department of CSE, 0114
9500 Gilman Dr.
La Jolla. CA 92093-0114

N-VERSION PROGRAMMING USING THE ADA TASKING MODEL

Don M. Coleman
Ronald J. Leach

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, DC 20059

ABSTRACT

This paper focuses on the use of the N-Version Programming (NVP) technique for achieving higher levels of software reliability in an Ada programming environment. Independent versions of an algorithm are implemented as independent Ada tasks; the Ada rendezvous model provides a natural mechanism for the sequencing and management of the NVP voting process. The paper describes the feasibility of using NVP to implement software fa̍ult-tolerance in an Ada multi-tasking envıronment.

1. INTRODUCTION

During the last few years, there has been increasing interest focused on program development methodologies that promote the production of reliable software. The traditional approach to reliability, "fault-avoidance", requires the software components and their integration techniques to be as nearly perfect (error-free) as possible. Efforts to enhance reliability include improvements in the design, testing, and other phases of the software development process. In spite of these improvements, experience shows that it is very difficult to produce error-free software. With this in mind, a body of research has been aimed at the development of techniques for "software fault-tolerance". These techniques use redundancy to provide an extra measure of reliability. Software fault-tolerance techniques, when used to complement the emerging improvements in fault-avoidance methodologies, appear to offer a promising approach to increased software reliability.

A pioneering effort called "N-Version Programming" (NVP) initiated during the 70's at the University of California at Los Angeles, was suggested as a possible technique for software fault-tolerance ([1], [3], [5]). The technique is based upon N-fold Modular Redundancy, which is a commonly used technique for hardware fault-tolerance. Another approach which received considerable attention around that time was called the Recovery Block Technique [7]. In this paper we do not consider the Recovery Block Technique.

N-version programming is defined as the independent generation of N (>1) software modules, called "versions", from the same initial specification. "Independent generation" here means that programming efforts are carried out by individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms and programming languages or translators are used in each effort. The intention with this method is to have as much independence as possible between the *N* different software implementations each of which constitutes a solution to the problem. The work of Knight and Leveson [6] raises some important points about insuring that the actual versions are independent.

A concurrent programming environment is a natural vehicle for the development of a *NVP* fault-tolerant software system. We have, with Ada, a programming language which includes concurrency as a direct construct within the language. This is provided through Ada tasking and the communication between tasks is done using the Ada rendezvous mechanism. Within an Ada program there may be a number of tasks, each of which has its own thread of control. Hence, it is possible to match the parallel nature of *NVP* with an Ada syntactical form which reflects the *NVP* structure.

In this paper we investigate the use of Ada as a tool for production of fault-tolerant software using *NVP*. In particular we perform an experiment to determine the overhead, measured in excess algorithm execution time, associated with the N-version implementation. The paper [4] provides a discussion about the feasibility of the use of C for *NVP* in a UNIX environment.

2. CONCURRENT PROGRAMMING

The basic notion in achieving fault-tolerance using *NVP* is to have more than one execution stream for the solution of the problem. That is, the problem is solved "concurrently" N independent times; results from these solutions are polled to determine the common or overall solution. In this sense *NVP* would seem to be well suited to concurrent programming.

Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and for solving the resulting synchronization and communication problems [2]. In general, a sequential program executes a list of statements in strict order or until modified by the appropriate control construct. A concurrent program specifies two or more threads of control (or sequential programs) which execute concurrently. In the Ada language, threads of control within a concurrent program are called tasks; in the UNIX environment they are known as processes. The actual execution of tasks and processes may be accomplished by a variety of methods. In cases where there are multiple cpu's, processes may execute simultaneously or in parallel; communication may be through a shared memory or a bus. In cases where there is a single cpu, each process must take its turn and it is called interleaved processing. In the discussion which follows, we have an *NVP* implementation in which N independent tasks are interleaved on a single cpu. Even though these tasks execute asynchronously, race conditions are not an issue as we use the Ada rendezvous for synchronization and control. Note that an Ada implememtation in a UNIX enviornment, such as the one that was used on this project, multiple Ada tasks are implemented within a single UNIX process and thus must share in the resources that are allocated to a single UNIX process.

The only mechanism provided in Ada for intertask communication is the rendezvous. As explained in the next section, the independent versions in a *NVP* scheme must communicate their results to an arbitrator (voter) which has the responsibility for synchronizing and selecting a "correct" solution from among the N independent processes. Each of the N versions provides an input for the "arbitration process" by a voter-initiated rendezvous. In a sense, one can consider the passing of parameters between the voter and a task similar to the passing of parameters between subprograms.

3. FRAMEWORK FOR THE NVP METHODOLOGY

The *NVP* methodology requires the polling of the independent processes at intermediate points of the algorithm called break points. At each break point the voter determines the "correct" state for the computation. The method for determination of the "correct" state can be as simple as a majority vote among the tasks. Hence, correctness is really a consistency check; i.e., the most frequently occurring value at a break point becomes the "correct" state. Further, it is assumed that differences between the tasks are not the result of errors in the voter. Therefore, the voter should be fault-tolerant or subject to a formal proof of correctness; or it should be developed from existing software components that are sufficiently tested to provide the highest level of reliability. The number of voting processes is an open question and is application specific; but for certain real-time programs it appears that the number will be small [4].

The impact of the selection of break points on the methodology is not well-understood at this time. Some ad-hoc rules may be inferred in specific instances. For example, any change to a fundamental data structure is an ideal candidate for a break point. The voter design includes strategies for dealing with different types of results; e.g., do numerical results have to be identical or simply within a given tolerance; how are character strings to be compared; what is the most appropriate technique for performing the minimal number of comparison tests between versions; what is the procedure for removing processes which gave consistently minority results; how to proceed in the case of catastrophic failure; and what to do when the number of consistent processes is below some minimal number.

The *NVP* scheme must be designed such that the exceptional termination of a particular version or task will not lead to the crashing of the total system. Otherwise, the fault-tolerance of the overall system may be decreased by addition of multiple versions. The approach is to minimize the coupling between the program modules. First, there is no coupling permitted between the various independent versions. Communication exists only between the main driver task and the voter or between the voter and the versions. In Ada, errors and other exceptional conditions for which exception handlers have not been provided cause the termination of the task in which the exception occurs. Hence, there is no propagation throughout the *NVP* system provided that the voter is designed to prevent any rendezvous with an aborted task. Furthermore, in Ada the proper use of the selective wait can provide a mechanism for protection and control of the "domino" impact of the failure of a single task. The voter can abort any task which it deems to be faulty. This determination is made when the task not being among the majority respondents some predetermined fraction of the time.

Protection against catastrophic failure due to an aborted version can be detected in the calling task because the aborted version will generate the predefined exception TASKING ERROR. Non-catastrophic errors can occur in numerical routines because of the inaccuracies of floating point arithmetic. As we shall see, this is not an issue here because we do an exact match with strings.

4. THE EXPERIMENT

The experiment examined *NVP* in the instance of a program designed to emulate a desk calculator that could perform arbitrary precision arithmetic. The versions of the algorithm was created by using modules encoded by different programmers. The desk calculators used a stack for the storage of intermediate results. The contents of the stack were pointers to structures pointing to arrays of characters that represented the values of the operands of the desk calculator. The operators and operands for this problem were given in data files with one operator or operand per line. Thus the size of the data file (as measured by the number of lines) indicates the number of activities of the stack since each operand is pushed onto the stack and each operator requires stack access. Each version of the algorithm was then modified to be an Ada task. The different versions of the algorithms were then modified to have each stack operation (push or pop) communicate the the value pushed or popped to the voter task. Thus each version of the algorithm was made into a task that communicated with the voter. Since the voter is another Ada task, communication is done using the rendezvous mechanism. A driver task controlled the entire system.

The development environment for this research was two UNIX machines running Ada compilers. The code was initially developed on an AT&T 3B2/500 with a preliminary version of AT&T Ada. This computer had 4 megabytes of main memory and ran AT&T System V version 3.1 UNIX. The versions of the algorithms were created by students in a very short period of time. The students had already implemented the same algorithms in C and thus the short amount of time for implementation did not appear to be unreasonable.

The original intent was to be able to use the same data files that were used in [4]. We ran into some problems when we had 6 versions of the algorithm, corresponding to a total of 8 tasks including the driver and voter tasks. The preliminary version of the Ada compiler on the 3B2 did not support the use of large data files; indeed, the largest file that we were able to use was only 24 lines long. For this reason, we transferred our work to a SUN 3/60 running Verdix Ada version 5.5. The code for the voter, main driver, and each of the versions was ported to the SUN 3/60 and the experiment was repeated there. Even on the SUN, we were unable to use the large data files that were used in the previous C/UNIX experiment when 6 versions (8 tasks) were run. We were able to run the program on input files as large as 10,000 lines on the SUN when running the program in single-user mode. On the SUN the executable file was 300 kilobytes and the core image was approximately 500 kilobytes. Files of size 10,000 lines or more caused problems with many tasks because the total limit per UNIX process was set to 4 megabytes on the SUN and each task was run as part of a UNIX task.

The relative execution times for the experiment for the AT&T 3B2 and the SUN 3/60 are given in tables 1 and 2, respectively.

Table 3 contains information of a different type. It indicates the percentage of times that the various numbers of versions agreed. This table is relevant to a discussion of the utility of *NVP* as a method for improving fault-tolerance. The data in all three of these tables will be analyzed in the next section.

5. ANALYSIS OF RESULTS

The original motivation for this paper was the measurement of the overhead of NVP when implemented in a modern, high-level language that supports concurrency. We had expected run times to be nearly linear as a function of the number of versions, with the fixed constant so small that run times should have been nearly proportional to the number of versions running as separate tasks, at least for large input files. What we found was quite different.

Adding another version had only a small effect on the relative execution time. Much of the computing time was spent in the overhead of the system for file access and for execution of the voter task. The cost of *N* versions is always much less than *N* times the cost of one version. As outlined in [4], the overhead is roughly defined as the additional time cost of running a *NVP* system as opposed to running a single version.

An implicit assumption about NVP is that the versions are independent; that is, errors are unrelated. Knight and Leveson ([6], [7]) have raised some questions about the validity of this assumption based on an empirical study. The possible errors in our experiment fall into one of three categories:

system errors caused by overloading the Ada run-time system

logical errors in the voter

errors in the individual versions that are detected by the voter.

The system limitation errors occurred when we exceeded the local storage of the Ada tasks. On one computer, we were limited to 24 lines of input (with 24 breakpoints) while on the other we could have as many as 15,000 lines (and thus 15,000 breakpoints) if we ran the program with the operating system configured in single user mode. This is far more restrictive than the results in a similar experiment performed in C using UNIX interprocess communication instead of the Ada rendezvous mechanism. The two experiments were run on the same computers and thus in the same UNIX environment. We expect that many of these system problems would be alleviated, but would not completely disappear, if some of the user-tunable operating system defaults were changed.

Errors in the voter did not occur during the experiment.

Errors in the versions occurred at various levels. It was generally the case that one of the versions was quite poor and that its presence caused a fairly high level of disagreements between the versions, at least on some files. It is clearly useful to eliminate bad versions early in the software reliability process. *NVP* can be applied to removal of versions, even if formal specifications of system behavior are not available.

Observations

1. Table 3 indicates that the *NVP* system will allow the program to proceed with some assurance of correctness in almost all cases. The percentage of times that the entire system was in a state of fairly high inconsistency was relatively low, especially for large input files.

2. For some of the tested input files, the method of *NVP* did not appear to be effective.

3. For this particular application, there is no need to execute more than three independent versions to have a 99% chance that the program can proceed with consistency.

4. The timing results are different from those reported in [4]. This is primarily due to the manner in which concurrency is implemented in C and Ada on UNIX systems. Concurrency in C programs in UNIX is performed by having the different threads of control running as different process, each of which is controlled by the cpu performing a context switch. This treatment of concurrency has a high operating system overhead. Concurrency of Ada programs in UNIX is performed by the Ada run time system within the context of a UNIX process. Hence the operating system overhead of Ada tasking under UNIX is much lower.

5. The Ada tasking model is particularly appropriate for embedded real-time systems that often have only a run time system and no intervening operating system. Thus the types of results described here are portable to non-UNIX environments.

6. A distributed or parallel environment is more appropriate for *NVP* than a sequential one.

7. In a real software development environment, programs are developed by teams. Since some of our versions were assembled by using different programmer's work, we were simulating a realistic development environment.

6. SUMMARY

N-version programming in Ada is relatively easy to understand as a high level construct. In a single cpu UNIX environment, the overhead of additional versions as tasks is not large because of the lack of context switching between processes. However, the system is quite fragile due to run-time limitations if there are many tasks or many versions. The situation can be improved considerably by running in single user mode.

ACKNOWLEDGEMENT

Research of Don M. Coleman was partially supported by the Federal Aviation Administration and the Naval Surface Warfare Center. Research of Ronald J. Leach was partially supported by the Army Research Office.

REFERENCES

1. A. Avizienis and J. P. J. Kelly, Design Fault-Tolerance by Design Diversity: Concepts and Experiments, UCLA Computer Science Department, Los Angeles, California 90024, USA

2. A. Burns, Concurrent Programming in Ada, Cambridge University Press, 1985, p18.

3. L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation", Digest of the 8th Annual International Conference on Fault-Tolerant Computing(FTCS-8), Toulouse, France, June 1978.

4. D. M. Coleman and R. J. Leach, Performance Issues in C Language Fault-Tolerant Software, Computer Languages, vol 14, No. 1 (1989), 1-9.

5. J. P. J. Kelly, Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach, Report No. CSD-820927, UCLA, Computer Science Department, September 1982.

6. J. C. Knight and N. G. Leveson, "An Experimental Analysis of the Assumption of Independence in Multi-Version programming", IEEE Trans. Softw. Eng., vol SE-12, no. 1 (January, 1986), 106-109.

7. B. Randell, System Structure for Software Fault Tolerance, IEEE Trans. Softw. Eng., SE-1(1975), 220-232.

	NUMBER OF VERSIONS					
File	one	two	three	four	five	six
1	1.00	1.17	1.36	1.53	1.76	2.06
2	1.00	1.39	1.64	1.78	1.91	1.90
3	1.00	1.18	1.41	2.06	1.74	1.92
4	1.00	0.95	1.20	1.29	1.75	1.60
5	1.00	1.08	1.31	1.41	1.56	1.76
6	1.00	1.27	1.30	1.49	1.75	1.82
7	1.00	1.23	1.59	1.67	1.44	1.75
8	1.00	1.18	1.29	1.71	1.55	1.69
9	1.00	0.79	0.94	1.34	1.09	1.24
10	1.00	1.14	1.42	1.50	1.69	1.88
11	1.00	1.17	1.01	1.21	1.40	1.58

Table I Relative Times for AT&T 3B2

	NUMBER OF VERSIONS					
File	one	two	three	four	five	six
1	1.00	1.14	1.32	1.45	1.56	1.70
2	1.00	1.16	1.28	1.41	1.53	1.66
3	1.00	1.13	1.24	1.37	1.52	1.67
4	1.00	1.00	0.80	0.60	1.00	0.80
5	1.00	1.13	1.30	1.40	1.53	1.68
6	1.00	1.11	1.22	1.33	1.67	1.78
7	1.00	1.17	1.26	1.39	1.57	1.70
8	1.00	1.09	1.27	1.36	1.55	1.82
9	1.00	1.11	1.22	1.44	1.67	1.89
10	1.00	1.22	1.33	1.44	1.44	1.78
11	1.00	0.75	0.75	1.00	1.00	1.00
12	1.00	1.15	1.29	1.42	1.56	1.71

Table 2 Times for S UN 3

WEIGHTED SUMMARY (ALL FILES)

Number of	NUMBER VERSIONS IN AGREEMENT WITH ANOTHER VERSION					
versions	none	two	three	four	five	six
2	21%	79%				
3	1%	9%	79%			
4		0%	9%	91%		
5			0%	7%	93%	
6				0%	9%	91%

Table 3 Percentages of Agreement

Concurrency in Ada and VHDL

Carl Schaefer

MITRE Corporation
McLean, Virginia

Abstract

Despite the fact that the VHSIC Hardware Description Language has been heavily influenced by Ada, the languages have very different models of concurrency. Areas in which the languages differ are: basic notion of time (real time of day versus simulated, elapsed time); the means by which execution of concurrent units is coordinated (control flow versus data flow); degree of determinism; and the manner in which conflicts are resolved.

Introduction

The VHSIC Hardware Description Language (VHDL) was developed under the auspices of the United States Department of Defense. The requirement was for a language which could be used in the acquisition process to specify the behavior of an electronic component, and could be used in the design process to specify the internal structure of the component. With descriptions of behavior and structure in a standardized language, the government reasoned, procurement of replacement parts, insertion of new technology into old systems, and design of new components of great complexity (100,000 gates) would be streamlined. Development of the language and tools began in 1983. The first public version of the language, so-called VHDL 7.2 was released in 1984, and a set of tools (analyzer, simulator, and library manager) was delivered to the government the following year. Subsequently, the Design Automation Standards Subcommittee (DASS) of the IEEE Computer Society took on the task of standardizing the language. IEEE Standard 1076 [1] appeared in 1987. Since that time the language has gained wide acceptance. At the 1989 Design Automation Conference, virtually all major players in the computer-aided engineering (CAE) market announced their intention to support VHDL. Since 1990, VHDL has become a major factor in the field of design synthesis (automatic derivation of a structural description from a behavioral description), even though VHDL was not originally required to support design synthesis.

While Ada and VHDL were designed to serve different purposes, the syntax and semantics of Ada have influenced the definition of VHDL from the earliest stages of the VHDL program. This influence is undoubtedly due as much to political concerns (both languages are sponsored by the Department of Defense) as it is due to the technical excellence of the design of Ada. Among the more important similarities are the following:

1. Data Types. VHDL's system of types is clearly derived from Ada's. Like Ada, VHDL distinguishes the concept of a type from that of a subtype; the type of a VHDL expression can be fully determined at compilation time. Like Ada, VHDL has user-defined enumeration, integer, floating point, array, record, and access types. There are, of course, differences: there is no VHDL parallel to Ada's type derivation; VHDL lacks variant record types, task types, and fixed point types; VHDL has physical types and file types; and VHDL ranges are characterized by direction (ascending or descending) as well as by bounds.

2. Operator Overloading. Like Ada, VHDL provides for overloading of enumeration literals, operator symbols, and subprogam names. The initial version of VHDL, VHDL 7.2, provided overloading only for enumeration literals. During the IEEE standardization process, enough reviewers thought that full operator overloading was valuable enough to include in the standardized language (VHDL 1076).

3. Scope and Visibility. VHDL's notions of declarative region, scope of declaration, direct visibility (including direct visibility achieved via "use" clauses), visibility by selection, hiding of declarations, and the mechanism for resolving overloading all derive directly from the corresponding Ada notions.

4. Distinction Between Interface and Body. Like Ada, VHDL distinguishes between a construct that represents an interface (and is therefore accessible to other constructs) and a construct that represents an implementation (and is therefore not accessible to other constructs). VHDL parallels Ada in distinguishing package declaration from package body and subprogram declaration from subprogram body. In addition, VHDL distinguishes the entity declaration (the external view of a hardware component) from an architecture body (the internal behavior or structure of the hardware component).

5. Sequential Control Structure. VHDL constructs for sequential control – if statement, case statement, loop statement, exit statement, return statement, and procedure call statement – are similar to the corresponding Ada statements in syntax and semantics.

6. Explicit Rules for Elaboration. Like Ada, VHDL has explicit rules governing the elaboration of declarations. While in the Ada LRM, the rules for elaborating a declaration are presented together with the syntax and semantics of the declaration, in the VHDL LRM, all elaboration rules are collected into one chapter.

7. Program Library. Like Ada, VHDL relies on the notion of a program library and has explicit order-of-compilation rules governing the order in which program units may be entered into the library. The following table show the correspondence between compilation units in Ada and in VHDL (in VHDL, subprogram declarations and subprogram bodies are not library units; the architecture body is a secondary unit):

Ada	VHDL
package declaration	package declaration
package body	package body
subprogram body	(subprogram body)
subprogram declaration	(subprogram declaration)
generic declaration	-
generic instantiation	-
subunit	-
-	entity declaration
-	architecture body
-	configuration declaration

Given the number of significant similarities between Ada and VHDL, it is instructive to compare their approaches to concurrency. Section 2 of this paper gives a brief overview of the dynamic semantics of VHDL. Section 3 discusses the notion of time in the two languages. Following this background, Section 4 discusses the major differences between concurrency in Ada and VHDL. Ada and VHDL versions of a simple client-server model are given in an appendix.

VHDL Dynamic Semantics

A VHDL model consists of a set of signals and a set of processes. A signal is a typed object that has a current value and zero or more future values, each future value projected to become current at a different time in the future. A process is a transform function; it reads current signal values and outputs future signal values. Processes are connected to each other by signals: the output from one process may be the input to other processes; a signal may be input to multiple processes and output from multiple processes.

Execution of a VHDL model consists of repeatedly applying the following cycle:

1. Determine the nearest time in the future at which any signal in the model has a new value (transaction) posted; call this time T. Advance the clock to T.

2. Update the current values of all signals having transactions at T. If this update involves changing the value of the signal (that is, if the new value is different from the old value) there is said to be an event on the signal.

3. Determine the set of all processes that are sensitive to any signal having an event at T. Run (in no particular order) all processes in this set; as a result of executing these processes, new future values may be posted on signals. Each process will run until it executes a wait statement.

Figure 1 is a graphic representation of the exection cycle. Execution of the model continues until there are no transactions posted to occur in the future on any signal. The model is then said to be quiescent. Consider the following very simple model:

```
signal S, P, Q : Bit := '0' ;

process
begin
  S <= '1' after 20 ns, '0' after 40 ns ;
  wait ;
end process ;

process
begin
  P <= S after 20 ns ;
  Q <= P after 10 ns ;
  wait on S, P ;
end process ;
```

The model would become quiescent at 70 nanoseconds (ns), at which time all three signals would have the value '0' and there would be no future transactions posted on any of the signals.

Time

Ada and VHDL have very different concepts of time. In Ada, time is "time of day"; it refers to time that can be synchronized with clocks in the real world. In VHDL, time is elapsed, simulated time. The basic types that describe time in Ada are defined in package Standard and in package Calendar. Duration is a fixed-point type defined in package Standard. Values of type Duration represent seconds of real time. An Ada implementation must allow representations of durations up to at least 86,400 seconds (the number of seconds in a day), and the smallest positive value of type Duration that can be represented (Duration'Small) must not be greater than 20 milliseconds. Type Time is defined to be a private type in package Calendar. The function Calendar.Clock returns the current time, which can be converted into year, month, day, and seconds by the procedure Calendar.Split. Year, month, and day are represented by subtypes of Standard.Integer and seconds are represented by Standard.Duration. There are no Ada constructs that require expressions of type Calendar.Time, and only one construct, the delay statement, requires an expression of type Standard.Duration.

Ada does not specify how the function Calendar.Clock obtains the time of day. In particular, it does not require that Standard.Duration'Small be the same as System.Tick, the basic clock frequency of the computer system. However, it is clearly the intention that time in Ada is, in some sense, "real-world" time. Thus (if we ignore pathological cases in which an Ada program resets the system clock by means of a system call) the execution of an Ada program cannot influence the value returned by a call to Calendar.Clock, and the Ada language does not define any way in which the execution of an Ada program can determine the magnitude of the difference between two calls to Calendar.Clock. Furthermore, if two Ada programs running on the same processor could simultaneoulsy call Calendar.Clock, the two calls would return exactly the same value; previous execution history (including the time at which execution started) would have no effect.

In VHDL, type Time is defined in package Standard as a physical type with implementation-defined bounds. A VHDL physical type is a scalar type; a value of a physical type can be represented with a numeric literal followed by an identifier denoting a unit of measurement. In the physical type definition, each unit of measurement is defined to be some integral multiple of a base unit. The base unit of Standard.Time is the femtosecond (10E-15 seconds), whose unit identifier is "fs". Other units of type Time are the picosecond (ps), nanosecond (ns), microsecond (us), millisecond (ms), second (sec), minute (min), and hour (hr). Function Standard.Now returns the current (simulation) time in femtoseconds. Several VHDL constructs require expressions of type Standard.Time; the two most important uses are in waveform elements in signal assignment statements (to specify the delay before a projected value becomes the current value of a signal) and in timeout clauses in wait statements (to specify the maximum period of time that a process will be suspended).

```
– in waveform elements
Signal_1 <= '0' after 5 ns, '1' after 10 ns ;

– in a timeout clause
wait for 100 ns ;
```

Other contexts requiring the use of time expressions are the value of certain attributes defined for signals (for example, S'LAST_EVENT, which gives the amount of time that has elapsed since the last event occurred on signal S), the parameter for certain attributes defined for signals (for example, S'STABLE(10 ns), which returns a boolean value indicating whether there has been an event on S in the last 10 ns), and in disconnection specifications (to specify the period of time between the transition of a guard expression from True to False and the disconnection of drivers guarded by the expression).

In all contexts, time in VHDL is elapsed time. Standard.Now returns the number of femtoseconds elapsed since the start of model execution; at the start of execution of any VHDL model, a call to Standard.Now returns the value 0 fs. In those constructs that require expressions of type Standard.Time, values are elapsed time relative to the value of Standard.Now at the point at which the expression is evaluated. For example, the signal assignment statement

```
S <= '1' after 20 ns ;
```

specifies that after 20 ns in the future, '1' will become the value of signal S. And the wait statement

```
wait for 100 ns ;
```

specifies that the process containing this wait statement will be suspended for the next 100 ns.

Time in VHDL is simulated time; it has no relation to real time. Unlike Calendar.Clock in Ada, Standard.Now in VHDL is determined by the execution of the VHDL model. Furthermore, the rate at which VHDL Standard.Now advances with respect to wall-clock time in the real world is not constant. If the model is executing a VHDL process statement that lacks a wait statement, then the value of Standard.Now will never advance regardless of how long (by wall-clock time) the model executes. On the other hand, it is possible for an executing model to advance the value of Standard.Now by 1 hr (one hour) in one millisecond of wall-clock time.

Figures 2 and 3 contrast VHDL's and Ada's notions of time and concurrency.

The simulation cycle described above is complicated somewhat by the possibility of zero-delay signal assignments in the model. A signal assignment always posts a *future* transaction, even if the time expression evaluates to zero. Consider the following VHDL process:

```
process
begin
  S <= P after 20 ns ;
   S_last_val <= S after 20 ns ;
  wait on P ;
end process ;
```

Suppose that this process executes at time T. At the point of the wait statement, the process has posted two future transactions, both to take effect at time T+20ns. The value of S_last_val at time T+20ns will be equal to the value of S at T, and the value of S at T+20ns will be equal to the value of P at T. Suppose that the process were rewritten as follows:

```
process
begin
  S <= P after 0 ns ;
   S_last_val <= S after 0 ns ;
  wait on P ;
end process ;
```

The signal assignment statements now specify that the transactions are to be posted after a zero-time delay. Nevertheless, the transactions do not take effect immediately; they take place at some time in the future that is still earlier than one femtosecond (the minimum measurable duration in VHDL) in the future. Therefore, at the point of the wait statement, the values of S and S_last_val will not, in general, be the same. A transaction that is specified with a time expression that evaluates to zero is said to take effect after one "delta delay". Delta delays are infinitesimal: no number of them will add up to the miminum measurable duration of one femtosecond. However, a delta delay will induce an additional simulation cycle. A delta cycle proceeds in the same fashion as does a normal cycle except the clock is not advanced. First, all signals having a delta-delayed transactions are identified; the new values are made current values (without advancing the clock); those signals having transactions that are events are identified, and all processes sensitive to these signals are run. As a result of running these processes, additional delta cycles may be induced. In fact, it is possible to specify a process that will induce an infinite number of delta cycles, preventing the clock from ever advancing:

```
process
begin
  S <= not S after 0 ns ;
  wait on S ;
end process ;
```

Clearly this is not a model of a useful piece of hardware; nevertheless, the semantics of VHDL allow this just as the semantics of Ada allow inifinite loops and deadlocking tasks.

Concurrency

Having examined the dynamic semantics of VHDL and the concept of time in Ada and VHDL, we turn to the notion of concurrency. Ada and VHDL are both "concurrent" languages, but they have very different conceptions of concurrency. In Ada, the basic unit of concurrency is a declared or allocated object containing entries that can be called; in VHDL, the basic unit of concurrency is the process statement, which cannot be referenced or called from outside. Ada and VHDL have different methods of coordinating execution across the basic units of concurrency. Ada programs are nondeterministic while VHDL programs are deterministic. Finally, Ada and VHDL have different methods of resolving conflicts.

Coordination of Execution.

The basic unit of concurrency in Ada is the task. A task is an object that is declared or allocated in a package, subprogram, or another task. Like other objects declared in packages, a task can be made visible outside the package declaration in which it is declared, and tasks can be passed to subprogam parameters. Anywhere a task is visible, its entries can be made visible by selection. The basic unit of concurrency in VHDL is the process, which is defined by a process statement in an architecture body. A process may be associated with an identifying label, but the process is not visible to any other processes, neither to sibling processes contained in the same architecture body nor to processes contained in other architecture bodies. Unlike the Ada task, the process is not an object that can be passed to subprogram.

In VHDL, the number of processes in a model is determined by the elaboration of all the architecture bodies in the model, which occurs before the execution of any VHDL statement in the model and before the simulation clock advances beyond 0 fs. As part of the first simluation cycle, every process in the model is activated, in no particular order. The number of processes in a VHDL model remains constant for the duration of the model simulation; VHDL does not allow dynamic creation of processes and does not define process completion or termination (although a VHDL process that executes an unconditional wait might be thought of as having completed its execution). This static scheme of process creation and destruction makes sense in a language that is describing hardware: hardware components do not dynamically appear and disappear. In contrast, an Ada task is not constrained to model some physical entity. To account for the possibility of dynamic creation and termination of tasks, Ada has a complex set or rules governing the activation and termination of tasks. There is also a difference, by several orders of magnitude, between the number of tasks in a typical Ada program and the number of processes in a typical VHDL description. A low-level VHDL description of a large VHSIC device might have 100,000 processes. As a practical matter, implementing a simulator capable of computing 10,000 transactions per second on a model with 100,000 processes and an equal number of signals would be very difficult if dynamic creation of processes were allowed.

Perhaps the most fundamental difference between Ada and VHDL is in the means by which they achieve coordination among concurrent units. Ada tasks coordinate their execution through entry calls; in this sense, Ada is a control flow language. In contrast, VHDL (at least the concurrent sublanguage of VHDL) is a dataflow language. A VHDL process is not called or invoked; there is no flow of control among VHDL processes. Real hardware consists of a number of components that are continuously "active" or "executing" in the sense that they are continuously supplying some signal level to each of their outputs. To achieve some efficiency, VHDL models an abstraction of real hardware. In this abstraction, each process only needs to be active when one of its outputs could change, and this can happen only when one or more of its inputs has changed. Thus, a VHDL process is active only in a simulation cycle in which one of the signals to which it is sensitive (a subset of the set of signals the process can read) has a change in value. The set of signals that the process is sensitive to may change from one simulation cycle to the next; however, the current sensitivity set is always a subset of a set that is known at the start of execution (after the model has been completely elaborated).

An Ada task waits for an execution event -- a called task completes execution of an accept statement, or a calling task executes an entry call statement -- but cannot wait for a data event. A VHDL process waits for a data event -- a change in value on a signal -- but cannot wait to be called directly by another process. Ada tasks and VHDL processes can both wait for timeouts events -- the delay statements in Ada, the for clause in a VHDL wait statement. However, in Ada, the use of delay statements is not a reliable way to coordinate execution across tasks.

Deterministic Execution.

Two Ada tasks run concurrently in the sense that, in the general case, Ada does not specify whether a particular statement in one task will execute before or after a particular statement in the other task. Order of execution is deterministic only if one task executes a call to an entry in the other task, and in this case it is guaranteed only that the calling task will not execute the statement following the entry call until the called task has finished executing the sequence of statements in its corresponding accept statement.

Ada also does not allow the programmer to specify when a particular statement will execute with respect to time. A program may contain the delay statement

```
    delay Time_to_wake_up - Clock ;
```

However, Ada does not guarantee that the task will resume execution at the intended time of day; it guarantees only that the calling task will be suspended for at least the amount of time specifed in the duration expression.

There are four ways in which Ada is nondeterministic. First, as just mentioned, Ada does not specify an upper bound on how long a task that has executed a delay statement will remain suspended. Second, Ada does not specify which of several ready tasks of equal priority will run next. Third, Ada does not specify which of several open accept alternatives in a selective wait statement will be chosen. Fourth, Ada does not specify the order in which several tasks declared in the same declarative region will be activated. This nondeterminism does not make Ada a defective language. However, it does mean that a legal Ada program may not have the same effect every time it is run. A program may be run on two different (single-processor or multi-processor) systems, and allowed to terminate normally in each instance, but may have different final states; that is, the static variables may have different values at the end of the two executions. In fact, if tasks in the program execute delay statements, two executions of the program on the same system may result in different final states.

In contrast, VHDL programs are deterministic. If a process specifies a certain delay, the process is guaranteed to resume after precisely (as measured by Standard.Now) the specified amount of time. Every time a VHDL program is run -- whether on a multi-processor system or on a single-processor system, whether on one machine or on several -- the program will result in exactly the same state. (Strictly speaking, this is true only if the model runs to quiescence; if a process were to execute an assertion violation that caused the model to terminate before quiescence, there is nothing in the language definition to guarantee that successive runs of the model would result in the same final states.) Hardware designers require a high degree of determinism in a simulation language like VHDL. A designer is interested in monitoring not only the values of the relatively few signals connected to the environment but also the values of any of the signals internal to the model at any point in the simulation to detect violations of timing constraints, underdriven busses, etc. The determinism of VHDL insures that such conditions are repeatable from one simulation run to the next.

Resolution of Conflicts.

Two concurrent units can come into conflict either because they simultaneously require the service provided by a third unit or because they require simultaneous access to a data item where at least one of the accesses is an update.

In VHDL it is not possible for two processes to conflict over access to a service provided by a third process since, as explained above, one VHDL process cannot invoke another. In Ada, it is possible for two tasks to require the services of a third task, in which case the conflict is resolved by FCFS queuing at the task entry.

Conflicts over access to data items can occur in both languages. Given that it is not a data flow language, it is not essential that Ada define a mechanism for resolving conflicting accesses to a data item, and indeed it does not define any such mechanism. VHDL handles conflicting data accesses differently, depending on whether the conflict is read-write or write-write. Strictly speaking, read-write conflicts are excluded by the language since the current value of a signal is determined at the start of a simulation cycle and cannot change within a simulation cycle. A write to a signal is always a write that projects a value at some time in the future, even if only one delta in the future. And since time does not advance, even by one delta, within a simulation cycle, there is only one well-defined value for any signal within a simulation cycle. Write-write conflicts are more complicated. It is possible for each of two processes to put a transaction with the same (future) time component but differing value components on a single signal. However, the semantics of the language require that any such signal, called a resolved signal, have an associated user-defined resolution function. Such a resolution function has a single parameter whose type must be an unconstrained array having the type of the resolved signal as its element type; the values passed as the N elements of the array parameter are the N current values projected by the N processes that execute signals assignments to the signal (called the "drivers" of the resolved signal). The value of a resolved signal is automatically re-computed by the associated resolution function at the start of each simulation cycle in which any of the resolved signal's drivers has an event. A typical resolution function might compute the "wired or" or "wired and" of the driving values.

Conclusion

Both Ada and VHDL provide for concurrent computation, but their approaches to concurrency as well as their underlying notions of time are fundamentally different. Ada's time is the real-world time of day while VHDL's time is elapsed time of simulation. In Ada, the execution of concurrent units is coordinated through a network of entry calls and accept statements, while in VHDL coordination is through a network of data paths. These differences in the conception of time and in the notion of concurrency reflect the different functions of the two languages. As a real-time programming language, Ada must assist the programmer in coordinating the flow of control in algorithms with typically coarse-grained parallelism. As a hardware description and simulaton language, VHDL must assist the hardware designer in capturing the massively parallel data flow of hardware.

References

[1] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard VHDL Language Reference Manual* (IEEE Std 1076-1987). New York, 1988.

[2] R. Lipsett, C. Schaefer, C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, Boston, 1989.

Appendix

This appendix contrasts the approaches to concurrency in Ada and VHDL by means of a simple model consisting of a server and two clients. First the skeletal Ada is given. The program declares a task type for the client and a task type for the server. The server has one entry call with two parameters, one for the client's request and one for the server's reply.

```
package Example_types is
  subtype Request_type is Integer ;
  subtype Reply_type is Integer ;
end Example_types ;
```

```
with Example_types ; use  Example_types ;
procedure Example is
 task type Client_type ;
 task type Server_type is
  entry Request
   (Request_line : in Request_type ;
    Reply_line   : out Reply_type) ;
 end Server_type ;
 Client_1, Client_2 : Client_type ;
 Server_0        : Server_type ;

 task body Client_type is
  Request : Request_type ;
  Reply   : Reply_type ;
 begin
  loop
   – ...
   – formulate request in variable Request
   – ...
   Server_0.Request (Request, Reply) ;
   case Reply is
    – ...
   end case ;
  end loop ;
 end Client_type ;

 task body Server_type is
  Reply : Reply_type ;
 begin
  loop
   select
    accept Request
     (Request_line : in Request_type ;
      Reply_line   : out Reply_type) do
     – ...
     – formulate reply in variable Reply
     – ...
     Reply_line := Reply ;
    end Request ;
   or
    terminate ;
   end select ;
  end loop ;
 end Server_type ;

begin
 – ...
end Example ;
```

The VHDL equivalent will be given in two versions. In the first version, the model consists of a single entity declaration and a corresponding architecture body. The architecture body contains three processes, one for each of the two clients and one for the server. Like the Ada server task, the VHDL server process will take requests from the two clients in any order. Where the Ada had only one entry with two parameters, the VHDL has two pairs of signals. This is necessary since it is possible for two requests to arrive at exactly the same time and the VHDL model will not queue requests from multiple processes as the Ada will queue entry calls from multiple tasks. It would be possible to implement the design using a single pair of signals (or even a single signal for both requests and replies); this would correspond to a hardware bus and would require an associated bus protocol and resolution function. Notice that VHDL does not have a notion of a process type like Ada's task type. The second version of the VHDL model shows how the process code can be encapsulated in an architecture body and reused via component instantiation statements.

```
package Example_types is
 subtype Request_type is Integer ;
 subtype Reply_type is Integer ;
end Example_types ;

use Work.Example_types.all ;
entity Example is
end Example ;

architecture Example of Example is
 signal Request_1, Request_2 : Request_type ;
 signal Reply_1, Reply_2     : Reply_type ;
 constant Request_delay : Time := ... ;
 constant Reply_delay   : Time := ... ;
begin

 Client_1 : process
  variable Request : Request_type ;
 begin
  – ...
  – formulate request in variable Request
  – ...
  Request_1 <= Request after Request_delay ;
  wait on Reply_1 ;
  case Reply_1 is
   – ...
  end case ;
 end process ;

 Client_2 : process
  variable Request : Request_type ;
 begin
  – ...
  – formulate request in variable Request
  – ..
  Request_2 <= Request after Request_delay ;
  wait on Reply_2 ;
  case Reply_2 is
   – ...
  end case ;
 end process ;

 Server : process
  variable Reply : Reply_type ;
 begin
  wait on Request_1, Request_2 ;
  – reply to one or both of the request lines
  if Request_1'Active then
   – formulate reply
   Reply_1 <= Reply after Reply_delay ;
  end if ;
  if Request_2'Active then
   – formulate reply
   Reply_2 <= Reply after Reply_delay ;
  end if ;
 end process ;

end Example ;
```

The second VHDL version encapsulates the client process and the server process in separate entity-architecture pairs. These Client and Server components are then instantiated in the top-level Example architecture.

```
use Work.Example_types.all ;
entity Example is
end Example ;

use Work.Example_types.all ;
entity Client is
 port (Request_line : out Request_type ;
     Reply_line   : in Reply_type) ;
end Client ;

architecture Client of Client is
 constant Request_delay : Time := ... ;
begin
 Client_proces : process
  variable Request : Request_type ;
 begin
  – ...
  – formulate request in variable Request
  – ...
  Request_line <= Request after Request_delay ;
  wait on Reply_line ;
  case Reply_line is
   – ...
  end case ;
 end process ;
end Client ;

use Work.Example_types.all ;
entity Server is
 port
  (Request_line_1, Request_line_2 :
   in Request_type ;
  Reply_line_1, Reply_line_2     :
   out Reply_type) ;
end Server ;

architecture Server of Server is
 constant Reply_delay : Time := ... ;
begin
 Server_process : process
  variable Reply : Reply_type ;
 begin
  wait on Request_line_1, Request_line_2 ;
  – reply to one or both of the request lines
  if Request_line_1'Active then
   – formulate reply
   Reply_line_1 <= Reply after Reply_delay ;
  end if ;
  if Request_line_2'Active then
   – formulate reply
   Reply_line_2 <= Reply after Reply_delay ;
  end if ;
 end process ;
end Server ;
```

```
architecture Example of Example is
 signal Request_1, Request_2 : Request_type ;
 signal Reply_1, Reply_2     : Reply_type ;
 component Client
  port (Request_line : out Request_type ;
       Reply_line   : in Reply_type) ;
 end component ;
 component Server
  port
    (Request_line_1, Request_line_2 :
     in Request_type ;
    Reply_line_1, Reply_line_2    :
     out Reply_type) ;
 end component ;
 for all : Client use entity Work.Client (Client) ;
 for all : Server use entity Work.Server (Server) ;
begin
 Client_1 : Client port map
   (Request_1, Reply_1) ;
 Client_2 : Client port map
   (Request_2, Reply_2) ;
 Server_0 : Server port map
   (Request_1, Request_2, Reply_1, Reply_2) ;
end Example ;
```

Author

Carl Schaefer is a Lead Scientist with the MITRE Corporation in McLean, Virginia. Previously he was with Intermetrics, Inc., where he was engineering manager for Intermetrics' VHDL tools.

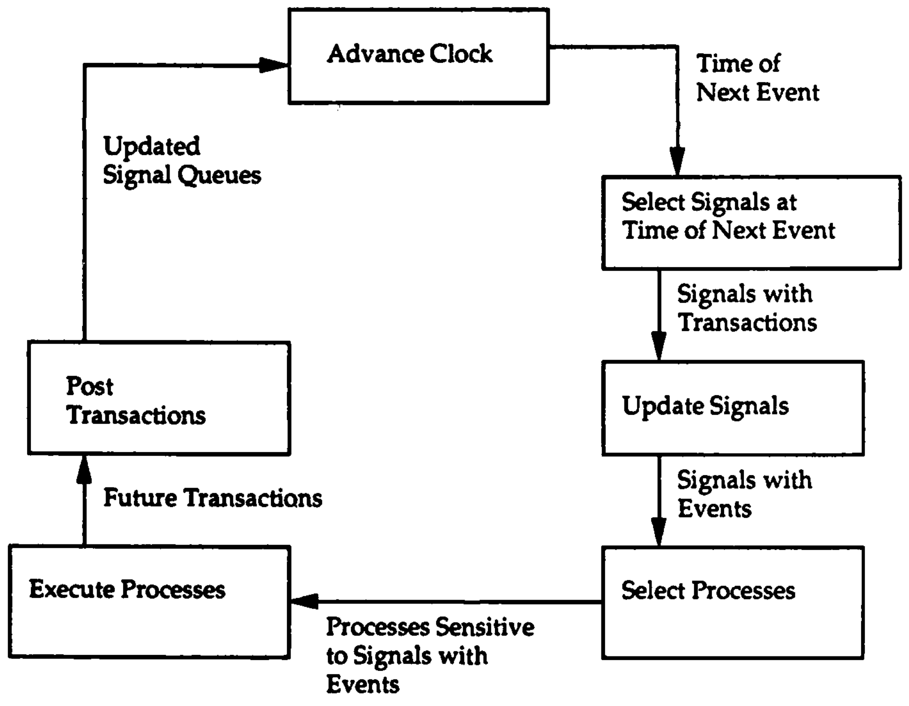
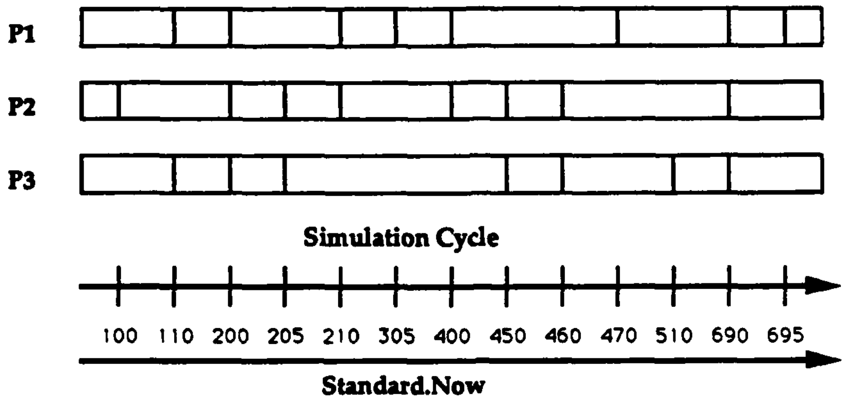
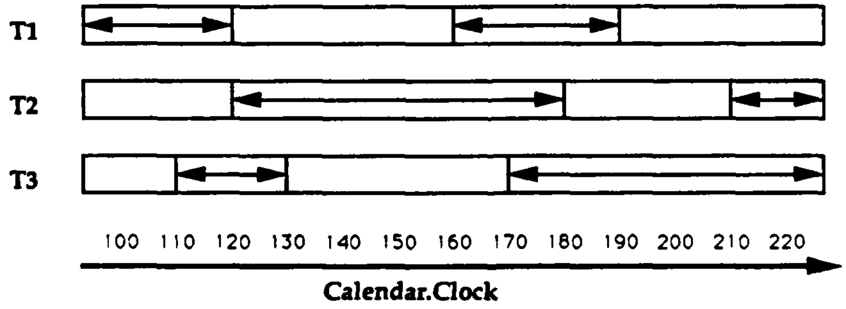


Figure 1: VHDL Simulation Cycle



Processes P1, P2, and P3 are concurrent, but not all of them are active at each simulation cycle. The simulation clock (Standard.Now) does not advance by the same amount of time for each new simulation cycle, and the clock does not advance during a cycle.

Figure 2 : VHDL Concurrency



Tasks T1, T2, and T3 are concurrent, but they are not all simultaneously active. Time (Calendar.Clock) advances at a constant rate regardless of how many tasks are active.

Figure 3 : Ada Concurrency

DEVELOPING INTERFACE STANDARDS FOR Ada SYSTEMS

Ella L. Krous

TELOS Systems Group
Lawton, Oklahoma

ABSTRACT

This paper presents a method for resolving a growing need of the Department of Defense (DoD) to develop and maintain standards for documenting the communication among software systems developed in Ada. These documented interface standards will become the design to documents, test to documents, and a management tool for developing Ada systems.

INTRODUCTION

These documented interface standards were initiated in December of 1988. The objective was to establish a common database which would provide support for the generation of interoperability specifications for selected Command and Control Systems. This paper will discuss the development of the data to be required in the interface standard for the developed Ada systems. The paper will also discuss the management of the interface information. Finally the paper will discuss how the interface information and resulting documents will be used to develop interfacing and interoperating Ada language systems to ensure interoperability among the fire support tactical data systems.

DEVELOPMENT OF MANAGEABLE DATA

There are various systems currently in the process of being designed and engineered to use Ada, where the target system is the Army's common nardware/software. The data required to create the interface specifications was taken from existing interface documents. The type of data required to develop the interface specifications was a definition of the interface identification, operational facility descriptions, system descriptions, communication protocol, message protocol(s), the messages, and the data element dictionary. The data has been used to create over 121 operational facility interfaces required for each of the Ada systems to interoperate and to interface in a tactical environment.

The database was constructed from the data required for transmission and reception on an interface. The pieces of data that are transmitted and received have been named data items. These data items were then grouped together according to usage, containing the maximum range values for the interface on which the information is used. The grouping of data by how it is used defines the data use identifiers. The data use identifiers were then grouped together and given a generic data field name. The data items, data use identifiers, and data field names were organized into a dictionary. The database can be searched to find information by generic data field and data use identifier.

For example, consider the mathematical model of a point defined in the three dimensional field along an X and Y axis. The point may be at (X,Y). The graph may be defined between X and Y coordinates of 0-100 and 0-50 respectively. The data items are entries X and Y may assume. The data use identifiers would be "Point Location, X" and "Point Location, Y". The generic data fields' names may be "Horizontal Axis Identifiers" and "Vertical Axis Identifiers". All coordinates using the same entries can now use the same data items. This allows the data to be entered once and used as many times as necessary. This commonality maximizes the data elements sent between nodes.

The database has now been created with specific information that will define the domain for messages to be created and exchanged among the nodes. The structure of the message is defined along with the message purpose, communication line description, set description, message compaction description, cases, conditionalities, and any special considerations required for usage by the developer. Once the information is stored in the database, the message map can be used between any two nodes.

The case statements developed for each message are to support the field operational requirements for a message to be on an interface. These are the minimum requirements for a message to achieve a specific purpose and still pass the receiving system's input syntax. The cases are related to the message they will be used to support. The cases are in an Ada format and use the same data elements as are on the message maps.

Conditionalities between data sets on a message are defined and produced in an Ada condition format. The actual data items that cross the interface are also monitored by conditionalities. This assists the implementation process for an Ada system. The conditions are stored in the database and are also related to the message they will be used to support. The conditions use the same data elements as are on the message maps.

The descriptions of individual operational facilities are developed concurrently with the data element dictionary. Because an existing system's description can change as the requirements are defined, the system description is written generically. The echelon that a system supports on either end of an interface is now called an Operational Facility (OPFAC). The interface is not defined between systems, but between OPFACs (node to node). The OPFAC descriptions are stored in the database.

The database has additional information relating to communication protocols for an interface. The communication protocols are developed using the International Standards Organization (ISO) seven layer model. The message protocol descriptions are also stored in the database. There is also necessary transmitting and receiving information, legal combination tables, and other unique information related to a specific interface stored in the database.

The OPFAC descriptions, selected message maps, selected cases, selected conditionalities, the pieces of the data element dictionary used on the selected messages and other interface data are then combined to produce an interface specification. This information can be recalled and used to perform requirements and/or impact analysis.

MANAGING THE INTERFACE DATA

The database is managed and maintained by a system independent team of engineers. The engineers are responsible for the integrity of the database, the consistency of the data, and the operational stability of the interface documents generated. Maintaining the interfaces in this manner has improved the awareness of interoperability problems and helped in eliminating problems with design differences between the nodes. Using a team of engineers to maintain the database, the OPFAC's requirements are defined, analyzed, and potential interoperability problems are identified. This improves the semantic and syntactic meaning of a message on an interface.

For example one OPFAC has a data element range for a numeric entry of 0-99; the other OPFAC on the interface has a range for the same numeric entry of 0-72. The entries 73-99 cause the second OPFAC to error and interoperability has not been achieved. These differences are identified to the participating OPFAC's system engineers by the interoperability engineer(s). A solution is achieved before the systems are developed. This prevents having to solve the problem late in the life cycle of the development phase. It is well documented that the later a problem is discovered, the more expensive it is to fix.

The improved awareness of the data exchanged between nodes has led to an improvement in the interoperability among the systems. The data is no longer exchanged with only the processing requirements for two OPFACs to interface. The processing requirements for an OPFAC are met, but from a field operational view, the data exchanged is complete enough for the interfacing OPFACs to achieve interoperability and complete their mission.

For the difference between interfacing and interoperating there is a common example. "Meet me for lunch.", is a complete message, but not enough information to have two people have lunch together. "Meet me for lunch in the restaurant across the street at 12:00", is more information, and provides a higher confidence level of having a meeting.

USING THE INTERFACES

The resulting interface documents will be used as design to and test to documents for developing the Ada software. New systems to be developed in Ada will be able to reuse the tactical communication information in the documented interface. The selected message maps, selected cases, selected conditionalities, and the appropriate portions of the data element dictionary are used by the system development personnel designing and engineering an Ada system. The OPFAC system description may be changed as necessary when a completed Ada system replaces the existing field system. This allows reuse of the interface document.

The interface documents are also used by the system test specialists as test to documents. The new Ada systems should be able to operate in the same manner as an existing system. The data on an interface is clearly defined between the two OPFACs. The interoperability tests are accomplished in a similar manner to the unit tests. Each case and conditionality statement are now easily identified and tests can be written for each one.

Because a database of all the generated interface documents is maintained, future changes to a particular message can be used to develop new requirements. Changes to a particular message can also be used to predict impacts on current information processing and system development.

The results of analysis performed on the database will show how changes in one node's requirements will affect another node's requirements. These results can be used in resolving and managing interoperability problems. If the proposed enhancements are to be implemented by an OPFAC, the interface document will be used as a design to document by the OPFAC system developer and maintainer.

For example, a requirement given to OPFAC A may not be given to OPFAC B. OPFAC B should have received the requirement also. This may develop an interoperability problem. This can be identified by analysis on the database. The requirement can then be further analyzed for implementation.

CONCLUSION

As the DoD development budgets are reduced, it is not enough to have all systems interface with each other. The systems must also interoperate effectively. It is no longer sufficient for an interface to pass pieces of data between two OPFAC's. The systems must develop the capacity to exchange information in a meaningful fashion to accomplish an OPFAC's operational mission. One way to assure this is to develop and use the new interoperability design documents to increase the interfacing capabilities of the Ada systems and their interoperability with other systems.

ELLA L. KROUS

TELOS Systems Group
P. O. Box 33099
Ft Sill, Oklahoma 73503-0099

ELLA L. KROUS has a bachelors degree in mathematics and a bachelors degree in computer science. Ella has over 5 years programming experience in designing, developing, and implementing complex government software applications for Department of Defense Command and Control Systems.

A LAYERED ARCHITECTURE FOR DBMS INTERACTIONS WITH AN ADA APPLICATION

Amber M. McKay and Richard A. Pederson

GTE Government Systems
1700 Research Boulevard
Rockville, MD 20850

Summary

On a large project, our company was faced with using Ada in combination with a relational database management system (DBMS) to manage a very large database in a distributed on-line transaction processing (OLTP) environment. In addition to interacting with an extremely large and complex database, we had the challenge of using a new DBMS product that was untested for similar production applications.

Our objective was to design an architecture that would provide a fully functional interface between the Ada application code and the DBMS, yet still utilize the many advantages of the Ada language. In addition, we wanted an architecture that would be simple to implement, easy to maintain, and cause the minimum impact on the productivity of the software engineers designing and implementing the application.

Design Approach

Architectural Features.

The objectives outlined above were achieved by designing the architecture to incorporate the following features:

1. Hiding of Database From Application. Use of the "abstraction" concept of hiding the details of the database from the application so that the database application code could be modified without requiring recompilation of the calling software. Using this concept was particularly important for our application since, initially, we were using a beta version of the DBMS product and wanted to insulate the calling Ada software from potential problems with the DBMS or the database access library. Hiding the database from the application also permitted us to develop applications that were insulated from any location references required to process against the distributed database.

2. Asynchronous Processing. Use of Ada's notion of "concurrency" to allow processes to work asynchronously with the database. This Ada feature was of special significance since it freed up the application software to do other processing while the DBMS was processing transactions against the database.

Four-Layer Architecture.

Designing the application so that it would take advantage of the above features required a four-layer architecture. Figure 1 depicts the layered architecture, and the following sections provide a detailed description, including examples, of how this design was implemented. In addition, we briefly describe how using the layered architecture simplified the software development process and facilitated project staffing and training.

1. Layer One - The DBMS. The first layer consists of the DBMS including its schemas, views, integrity mechanisms (i.e., rules and triggers used to enforce data and referential integrity), and stored procedures (i.e., precompiled SQL data manipulation and query statements).

2. Layer Two - Encapsulating the DBMS Access Library. The second layer consists of an Ada package that encapsulates all of the DBMS access library routines required by the application software. This package also manages all of the concurrent connections to the database and the asynchronous accesses of the database. A record type was developed to store all of the information necessary about a connection to the database. Figure 2 shows how this record type is defined within the Ada package and the following paragraphs provide a detailed description of the record.

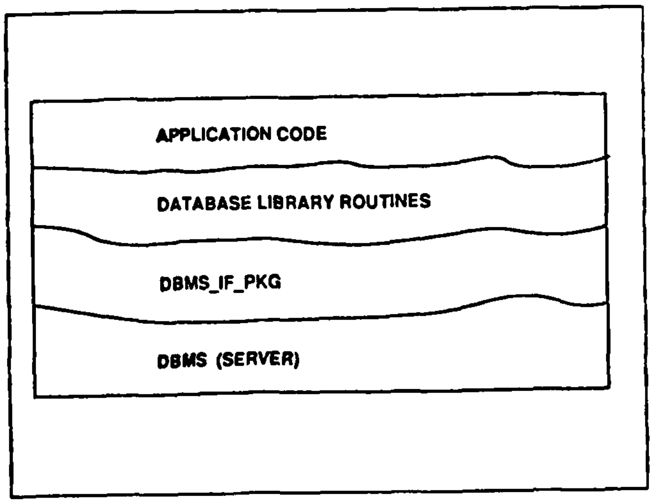


Figure 1. Layered Architecture for Database Accesses

```
typeDBPROCESS_REC is
  record
    TEMPORARY                    : BOOLEAN;
    DBPROC : DBMS_TYPES.DBPROCESS_PTR;
    DB_SYNC_TASK                 :
    DB_SYNC_TASK_PTR;
    ERRORS : ERROR_LIST
  end record;
```

Figure 2. Definition for Database Processing Record

The "TEMPORARY" boolean indicates whether the connection is a temporary or permanent connection. Processes that infrequently query the database make a temporary connection to the database prior to each query and terminate the connection immediately following execution of the query. Permanent connections are normally made at program startup and are not terminated until the program is shut down.

The "DBPROC" pointer is a structure that is maintained by the DBMS's access library. This pointer is used by the DBMS to differentiate between connections to the database server. Note: in a Client Server DBMS architecture, the Server is the portion of the DBMS responsible for managing accesses to and manipulation of the database; the Client portion of the DBMS is the interface to the user.

The "DB_SYNC_TASK_PTR" is a pointer to a task used as a blocking task. A blocking task prevents execution of an application task that is calling an access routine, thus allowing other application tasks to be scheduled. The database access routines send an asynchronous request to the database server and then wait for the task to receive notification of completion from the database server.

The "ERRORS" field is a pointer to a linked list of error messages. Routines are set up for message handling and for error handling. When a connection is made to the database server, these routines are assigned to process database messages or errors as they are generated. Each error message that occurs is attached to the linked list of messages. Later, when the application needs to process the errors, special routines are called to traverse the linked list of errors, interpret them, and send the errors back to the user application in a string variable or an enumerated message. Fatal system errors are written to an error log file and an exception is usually raised.

The only routines permitted to manipulate the DBPROCESS_REC structure are routines included in the second-layer package. This package hides all of its internal processing from the user application so that the user application only needs to know whether the connection to the database server is temporary or permanent.

3. Layer Three - The Database Transaction Library. The third layer is composed of a library of packages that contains specific database transactions for each of the applications. To simplify the interface to the transaction library, a standardized way for application software to interface with the database was developed. A template for the interface code was built and provided to each software engineer who was developing application code requiring access to the database. The engineer could update the template with code specific to her/his application. This procedure facilitated rapid code development, minimized coding errors, and simplified training since it limited the number of engineers developing code to directly interact with the database. Figure 3 provides an example of a procedure that establishes a temporary connection to the database (if a permanent connection is not already established), sends an SQL string or a stored procedure name and parameters to the DBMS for processing, retrieves results from the DBMS, and processes errors.

4. Layer Four - The Application Code. The fourth layer consists of the application code itself. The application code is responsible for making calls to the third layer routines and for establishing permanent connections to the database. In the previous example, when the application code needed to retrieve data from the database, a simple call, "RTRV_DATA(DBPROCESS_INDEX,DATA_REC);", was issued. Following the call, the application had the results available in the DATA_REC record for processing.

To facilitate application development, we created a formalized method of documenting database interactions. This method enabled developers to build the database schemas, views, and stored procedures more efficiently. The method required building compilable Ada packages, called DataBase TRansactions (DBTRs), which documented all transactions against the database, but were only used for reference purposes. Each DBTR referenced multiple DataBase Interface (DBI) routines, which also consisted of compilable Ada specifications. Each DBI was then translated into a stored procedure (i.e., a grouping of one or more SQL statements that were compiled and stored in the database by the database server). Every DBI contained all of the inputs, outputs, and error handling required for the associated stored procedure. The DBTRs and DBIs, in combination, were then used to build the third layer of routines that passed the data back and forth from the application to the database. Although the DBTRs and DBIs were designed to formally document the database interactions, they were created as compilable Ada specifications to ensure that all of the data types referenced actually existed. Figure 4 provides an example DBTR and Figure 5 provides an example DBI.

A Layered Approach - Observations

Benefits.

Overall, we found the layered approach, described within this paper, to be an extremely effective method of documenting, developing, and maintaining database interactions for a very large database application. The following benefits were derived from this approach.

1. Coding of Repetitive Accesses Facilitated. Due to the size of our application, many similar and/or repetitive database accesses were required. The layered approach minimized the effort required to code those accesses for the many different applications.

```
procedure RTRV_DATA (DBPROCESS_INDEX : in TYPES_PACKAGE.DBPROCESS_PTR;
                     DATA_REC : inout TYPES_PACKAGE.DATA_REC_TYPE) is

– PURPOSE:
–        The RTRV_DATA procedure queries the database using SQL commands to
–        retrieve <Enter purpose for retrieval>.
–
– PARAMETERS:
–        DATA_REC              – record to hold returned data
–        DBPROCESS_INDEX         – index into dbprocess_table (opt.)
–
– EXCEPTIONS:
–        DATA_BASE_ERROR         – raised when any error occurred during
–                             – the   database access
–
– METHOD:
–        The RTRV_DATA procedure sends a SQL command buffer to the
–        database server to retrieve <Enter method of retrieval here>.
–        This routine implements the RTRV_DATA_DBTR.
–
–––––––––––––––––––––––––––––––
–
  RESULT_CODE : DBMS_IF_PKG.RESULT_CODE_TYPE;
  ROW_STATUS :  DBMS_IF_PKG>ROW_STATUS_TYPE;

begin
  –[ Connect to the database server and, if connection is permanent,
  –[ just allocate a DB_SYNCHRONIZE task
  DBMS_IF_PKG.CONNECT(DBPROCESS_INDEX,"<Enter Database Server Name >");

  –[ Fill command buffer
  DBMS_IF_PKG.ADBCMD(DBPROCESS_INDEX,
    "<Enter Stored Procedure Name or SQL Command and Parameters>");

  –[ Send commands to the database server
  if DBMS_IF_PKG.ADBSQLEXEC(DBPROCESS_INDEX) = DBMS_IF_PKG.FAIL
    then
      raise DATA_BASE_ERROR;
  end if;

  –[ Loop for   each results command
  loop
    –[ Determine result code
    RESULT_CODE   := DBMS_IF_PKG.ABDRESULT(DBPROCESS_INDEX);

– Error messages are called here
    –[ This routine just returns an error string
    –[ Fatal errors will raise an exception
    DBMS_IF_PKG.SYS_ERR_STAT(DBPROCESS_INDEX,INFO_MSG);

    case RESULT_CODE is
      –[ Exit loop when all results have been processed
      when DBMS_IF_PKG.NO_MORE_RESULTS =>
        exit;

      when DBMS_IF_PKG.SUCCEED =>
        –[ If data is being returned, bind data being returned
        –[ from the database into local program variables
```

Figure 3. Database Transaction Library Template (1 of 2)

```
            DBMS_IF PKG.ADBBIND
                   (DBPROCESS => DBPROCESS_INDEX,
                      COLUMN => 1,
                      VARTYPE   => DBMS IF PKG.CHARBIND,
                      VARLEN => DATA RECCHAR PARAM1'LENGTH,
                      VARADDR   => DATA_REC.CHAR_PARAM1'ADDRESS);

            DBMS_IF PKG.ABDBIND
                   (DBPROCESS => DBPROCESS_INDEX,
                      COLUMN => 2,
                      VARTYPE   => DBMS IF PKG.INTBIND,
                      VARLEN => DATA RECINT PARAM1'LENGTH,
                      VARADDR   => DATA_REC.INT_PARAM1'ADDRESS);

            –[ Loop for each SQL row returned
            loop
               –[Retrieve row from database server
               ROW_STATUS := DBMS_IF_PKG.ABDNEXTROW(DBPROCESS_INDEX);

               case ROW_STATUS is

                  –[Exit   when all rows are returned
                  when DBMS_IF_PKG.NO_MORE_ROWS =>
                     exit;

                  –[ Do processing on regular rows returned
                  when DBMS IF PKG.REG ROW =>
                     –[ If   you are expecting more than one row
                     –[ to   be returned, special processing to
                     –[ put data in a list, file, etc. gets
                     –[ incorporated here.
                     null;

                  –[ Unexpected error, raise an exception
                  when others =>
                     raise DATA_BASE_ERROR;
               end case;
            end loop;

         –[ Unexpected error, raise an exception
         when others =>
            raise DATA_BASE_ERROR;
      end case;

   end loop;
   –[ Disconnect connection from database, if connection
   –[ is permanent, connection will not be disconnected.
   –[ The DB_SYNC_TASK is freed up to be used for another
   –[ connection.
   DBMS_IF_PKG.DISCONNECT(DBPROCESS_INDEX);

–[ Unexpected errors
exception
   when others =>
      DBMS IF PKG.DISCONNECT(DBPROCESS_INDEX);
      raise DATA_BASE_ERROR;

end RTRV_DATA;
```

Figure 3. Database Transaction Library Template (2 of 2)

```
_**********************************************************************
_**********************************************************************
_**
_**                DATABASE TRANSACTION SPECIFICATION
_**
_**          This  package is a design specification used by Database Engineering and S/W
_**          Development as a basis for agreement for the functions included in a
_**          database transaction. This will not be linked into an Ada executable.
_**
_**********************************************************************
_**********************************************************************
-
-    DBTRANSACTION NAME: RTRV_DATA_DBTR
-
-    CREATION INFORMATION: <Enter Date and Author>
-
-    REVISION HISTORY:
-         DATE     NAME        SUMMARY
-       <Enter revision information>
-
-    PURPOSE:      <Enter a high level purpose for the DBTR
-                    (Why is the data being retrieved and how
-                    does it fit into the big picture?)>
-
-    DESCRIPTION:
-       The RTRV_DATA_DBTR procedure calls the RTRV_DATA_DBI
-       to retrieve data. <Include application specific
-       details about the transaction.>
-
--      GENERAL   ADVISORY/ERROR CONDITIONS:
-       <Specify   error handling options, raise an exception,
-       return a   status, etc.>
-
-    NOTES:     <Any notes that would be helpful in documenting
-              the transaction are included here.>
-
-    CALLING LIBRARY PROCEDURE:
-       <Specify   package and calling procedure name>
-
-    DBDIST?       <YES/NO indicates whether this is a distributed update across servers>
-
-    DATABASE SERVER LOCATIONS: <Specify which database servers this
-                                 transaction will be applied against>
-
-    PERFORMANCE:    <Include any performance issues for reference>
-
-    FREQUENCY:   <Include frequency of queries and updates (low, medium, high)>
-
-    CONCURRENCY:    <Include any concurrency issues>
-
-    INTEGRITY:    <Include any data integrity issues>
-
with RTRV_DATA_DBI;

package RTRV_DATA_DBTR is

-       ***** Description of procedures that make up the transaction
-    DATASERVER_NAME : exec RTRV_DATA_DBI;

end RTRV_DATA_DBTR;
```

Figure 4. Example DBTR

```
-****************************************************************************
-****************************************************************************
-**
-**              DATABASE TRANSACTION SPECIFICATION
-**       This package is a design specification used by Database Engineering and
-**       S/W Development as a basis for agreement for the functions the SQL
-**       procedure is to perform.  This will not be linked into an Ada executable.
-**       It is written in Ada for convenience only and does not represent the
-**       actual   database library access.
-**
-****************************************************************************
-****************************************************************************
-
-    ABSTRACT:
-
-    APC NAME:  RTRV_DATA_DBI
-
-    KEYWORDS:   <Include any keywords>
-
-    CREATION INFORMATION:        <Enter Date and Author>
-
-    REVISION HISTORY:
-       DATE            NAME          SUMMARY
-          <Enter revision information>
-
-    PURPOSE:      <Enter a purpose for the DBI (why is the data being
-                   retrieved, updated, inserted and how does it fit into
-                   the   big picture?)>
-
-    METHOD:
-       DESCRIPTION:
-          <Enter a detailed description of the processing that needs
-             to occur>
-
-       SIDE EFFECTS:
-          <Enter any side effect information (e.g., a field is updated
-             that causes a trigger to perform some function - looking at
-             the stored procedure by itself would not necessarily show
-             that a   side effect will occur)>
-
-       GENERAL ADVISORY/ERROR CONDITIONS:
-          <Specify error handling options, raise an error, return a
-             status, etc.>
-
-       NOTES:      <Any notes that would be helpful in documenting the
-                   DBI   are included here.>
-
```

Figure 5. Example DBI (1 of 2)

```
--        CALLING LIBRARY PROCEDURE:
--            <Specify package and calling procedure name>
--
--        DATABASE  SERVER LOCATIONS:
--            <Specify which database servers this transaction will be applied against>
--
--        DBDIST?       <YES/NO indicates whether this is a distributed update across
--                    servers>
--
--        PERFORMANCE:                <Include any performance issues for reference>
--
--        FREQUENCY:                    <Include frequency of queries and updates (high,
--                                medium, low)>
--
--        CONCURRENCY:                <Include any concurrency issues>
--
--        INTEGRITY:       <Include any data integrity issues>
--
--    EXTERNALS:
--            Datatype packages:
--        <All data type packages are necessary to make sure they exist.>
--
_*******************************************************
package RTRV_DATA_DBI is

--        ***** INPUT_RECORD describes the order of the
--        ***** input parameters sent to the database

     type INPUT_RECORD is record
--    <Include all data types in parameter order for stored procedure>
--        null;
     end record;

     type OUTPUT_RECORD is record
--    <Include all data types in order of retrieval (for bind statements) that
--        the stored procedure needs to return>
     end record;

--        ***** DBGETMESSAGE_RECORD describes the error parameters being returned
--        ***** from the database

     type DBGETMESSAGE_RECORD is record
--    <Describe   any error messages that will be raised from the stored procedure
--        and the conditions that will cause them.>
        null;

     end record;
end RTRV_DATA_DBI;
```

Figure 5. Example DBI (2 of 2)

2. Training Requirements Reduced. The layered approach allowed coding of the actual calls to the database server access library to be made by a limited number of software/database engineers; therefore, only a small portion of the engineering staff required detailed knowledge of how to interact with the database server.

3. Database Server Problems Reduced. Since a new DBMS product was used, some problems occurred (due to bugs and lack of product knowledge). Because the applications were insulated from the DBMS, those problems had minimal impact on development of the applications.

4. Interfaces Standardized. By using the templates and standardizing both the documentation and implementation of the interfaces, short-term development costs and life-cycle maintenance costs have been reduced.

Conclusions

In addition to the benefits derived from the layered architecture described above, we came to the following conclusions:

1. It is critical that database interactions be documented as early as possible and that some formalized method of documenting the interactions (e.g., DBTRs and DBIs) be used. Without documentation, in large applications, numerous coding problems and inconsistencies are likely.

2. The layered approach becomes more beneficial as the size of the application increases. For very small applications, this approach could actually increase the amount of work required with little or no added benefit.

Authors

Amber M. McKay

GTE Government Systems
1700 Research Blvd
Rockville, Md 20850

Ms. McKay, a Software Engineer for GTE Government Systems, has spent nearly four years designing, developing, and implementing database intensive applications using high-level languages such as Ada. She has worked on a variety of projects ranging from small protoype applications to the very large database application described above. Ms. McKay is proficient in the use of the Ada and C programming languages, relational DBMSs, and SQL, the ANSI standard language supported by most relational DBMS products. She has a B.A. in Computer Science from the University of Maine.

Richard A. Pederson

GTE Government Systems
1700 Research Blvd
Rockville, Md 20850

Mr. Pederson, Manager of Information Engineering for GTE Governement Systems in Rockville, Maryland, has over 22 years of experience designing, developing, and implementing a wide range of computer applications for government and industry. He is familiar with a broad variety of database management products and programing languages, and has over four years of experience designing and developing database-intensive, Ada applications. Previously, at GTE, Mr. Pederson has served as Manager of Database Engineering and as software development manager. Prior to coming to GTE, he was manager for the Federal Government. Mr Pederson has a B.A. in English, with a minor in mathematics, from Clemson University.

TASKING FACILITIES IN ADA: "THE CIGARETTE SMOKERS PROBLEM"

Tina L. Newsome

Hampton University
Hampton, Virginia

This paper outlines the issues relevant in developing a student project which illustrates the use of parallelism to solve a problem using tasking facilities available in Ada. The problem addressed is "The Cigarette Smokers Problem" defined in chapter 9 of Operating System Concepts, by Peterson and Silberschatz. Synchronization and communication between the tasks used to solve this problem are facilitated thru entry calls and accept statements. Thus, thru the study of a simple problem relevant to concurrency such as this, insight can be made into solving real world problems such as real-time applications which involve concurrency control.

Concurrency is an issue which has evoked much interest because of the gains which occur as a result of exploiting concurrent (parallel) programming techniques. Concurrent, as defined in Webster's Third New International Dictionary, means "occurring, arising, or operating at the same time often in relationship, conjunction, association, or cooperation." This definition well encompasses the goal of concurrent processing; to allow naturally concurrent applications to be expressed as algorithms which execute simultaneously and which coordinate their activities efficiently through synchronized communication. Semaphores, critical regions, conditional critical regions, monitors, and message passing are approaches used for managing concurrency and solving the synchronization problems associated with concurrency.

This paper outlines the issues relevant in developing a student project which illustrates the use of parallelism to solve a problem using tasking facilities available in Ada. The problem, specifically "The Cigarette Smokers Problem", is defined in chapter nine of Operating System Concepts, by Peterson and Silberschatz, as follows:

> Consider a system with three smoker processes and one agent process. Each smoker continuously makes a cigarette and smokes it. But to make a cigarette, three ingredients are needed: tobacco, paper, and matches. One of the processes has paper, another tobacco, and the third has matches. The agent has an infinite supply of all three. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient can then make and smoke a cigarette, signaling the agent upon completion. The agent then puts out another two of the ingredients and the cycle repeats.

To enhance the problem, the following modifications were made to the above description:

1) To facilitate complete randomness, each smoker requests his ingredients from a distributor. The distributor distributes ingredients arbitrarily. This ingredient request makes it possible for the smokers to receive different ingredients for making and smoking cigarettes.

2) To allow the maximum amount of parallel processing to occur, upon receipt of the remaining two ingredients by a smoker, the agent immediately places another two ingredients on the table, allowing the other two smokers to make and smoke a cigarette if possible.

3) To facilitate complete fairness and randomness, the agent will never consecutively place the same two ingredients on the table for processing.

4) To prevent deadlock, the table ingredients are replaced after a significant number of unsuccessful rendezvous attempts

with the agent and no three processes will be distributed the same remaining ingredient.

The solution to this problem is implemented using the following six active tasks:

1) one id assignment task;

2) one agent task;

3) three smoker tasks;

4) and one distributor task.

The task ID_ASSIGNMENT assigns each smoker task a number used for identification purposes. The task AGENT requests ingredients to be placed on the table from the task DISTRIBUTE and accepts requests for the ingredients placed on the table from the smoker tasks. The agent only passes the ingredients if the requesting smoker has the remaining ingredient. The tasks SMOKER request an id number from task ID_ASSIGNMENT, requests the remaining ingredient from task DISTRIBUTE, and requests the table ingredients from task AGENT. The task DISTRIBUTE accepts requests from the agent and the smokers for ingredients. DISTRIBUTE simply calls a random number generator which generates integer numbers between zero and two, which correspond to the ingredients.

The synchronization and communication between tasks are facilitated through entry and accept statements. This message passing facility prohibits shared data objects and permits only the sharing of data values via parameter passing. Communication between two tasks occur when they rendezvous via the actual parameters in the entry call and the formal parameters in the corresponding accept statement. The task accepting the entry call causes suspension of the calling task until the information is exchanged; the suspension lasts until the execution of the accept statement is complete.

The tasks SMOKER request an id number by making an entry call to the task ID_ASSIGNMENT. ID_ASSIGNMENT is a simple accept statement enclosed within a for loop indexed from 1 to the number of smokers. The current for loop index is then passed back to the calling task as the id number via the parameter list. The smoker then continuously makes and smokes cigarettes by making a request for a remaining ingredient from the task DISTRIBUTE followed by a request for the table ingredients from task AGENT. These requests are also represented as entry calls to the corresponding tasks with the requested ingredients being passed thru the parameter lists. The making and smoking of cigarettes is simulated thru "delay" statements which suspend the execution of the calling task for a specified number of seconds.

The task AGENT infinitely loops making requests for table ingredients from task DISTRIBUTE via entry calls and accepting requests for those ingredients thru an accept statement.

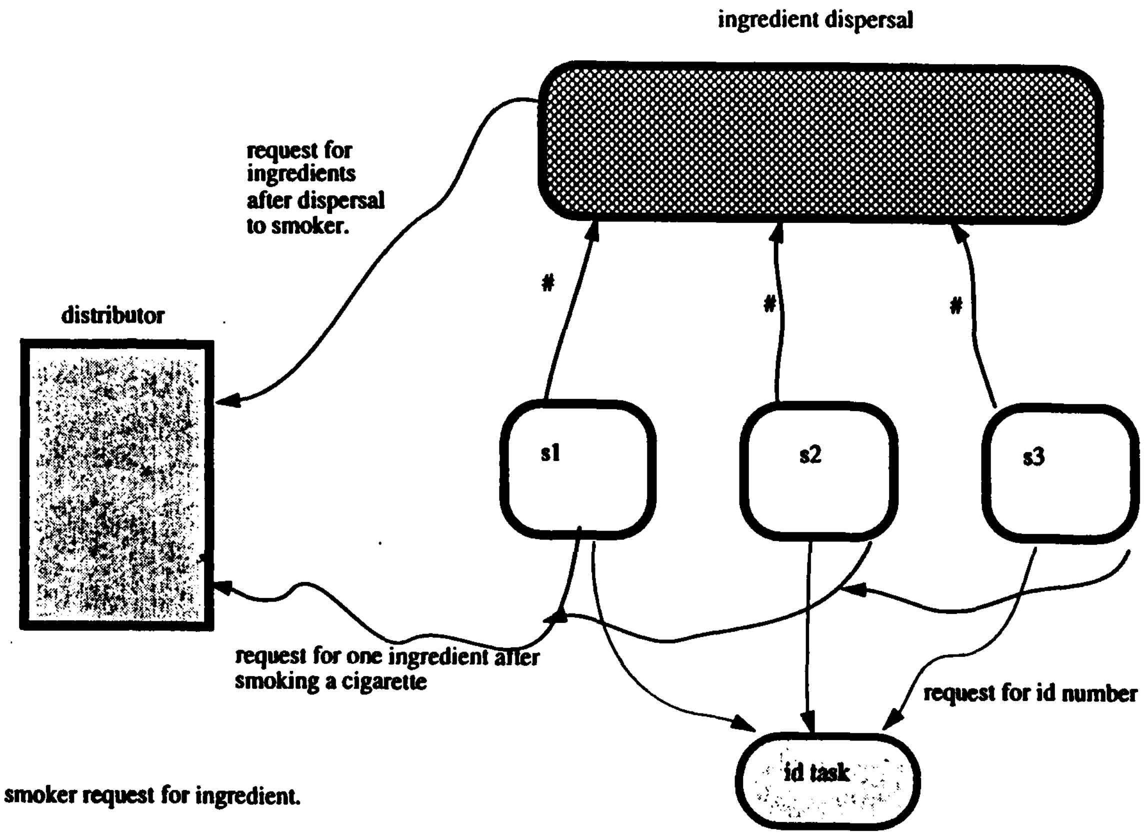
The task DISTRIBUTE infinitely loops accepting requests from the smokers and the agent thru accept statements. A select statement is used to distinguish the entry calls made by smokers from the entry calls made by the AGENT. DISTRIBUTE simply calls a random number generator to generate ingredients.

Mutually exclusive access must be provided to the id assigner and the agent when communication is necessary. When making the request for an id, the smoker must be guaranteed that the id it receives is unique. In addition, it must be guaranteed that requests for the table ingredients made by the smokers are mutually exclusive. Only one process should be able to access the table ingredients at a time. Thus, the implementation of the solution to this problem using entry calls and accept statements is ideal. Mutual exclusion to the critical sections and data exchange for process communication is provided by these mechanisms.

No major problems were encountered in trying to implement this algorithm mainly because of the facilities provided by Ada to guarantee mutual exclusion (a rendezvous mechanism which uses an implicit queuing scheme) and the enhancements added to the initial description of the problem to ensure randomness and fairness (which also helped to prevent deadlock).

In conclusion, the exploitation of concurrency has proven to be instrumental in gains toward implementing applications in parallel processing. As synchronization problems emerged, solutions to these problems were provided, each having its own advantages and disadvantages. These solutions include semaphores, critical regions, conditional critical regions, monitors, and message passing. Thus, various programming languages incorporate these mechanisms for process communication and synchronization allowing the structure of naturally concurrent algorithms to be elegantly expressed. More specifically, Ada, as illustrated in the solution to "The Cigarette Smokers Problem" implements message passing allowing process synchronization, as well as the exchange of data between processes for process communication. Thus, with its tasking facilities, Ada can be utilized to solve real world problems such as the bounded buffer problem and problems associated with real time applications which involve concurrency control.

Cigarette Smokers


ingredient dispersal
request for ingredients after dispersal to smoker.
distributor
#
#
#
s1
s2
s3
request for one ingredient after smoking a cigarette
request for id number
id task

smoker request for ingredient.

REFERENCES

1. Gehani, Narain. Ada: Concurrent Programming, pp. 14-26, 1984.

2. Peterson, James L. and Silberschatz, Abraham. Operating System Concepts, Addison-Wesley Publishing, Massachusetts, 1985, pp. 366.

Tina L. Newsome was born on October 29, 1968 in Lanham, Maryland. She received the B.S. degree in computer science from Hampton University, Hampton, Virginia, in 1990. She joined AT & T Bell Laboratories, Middletown, NJ, as a Member of Technical Staff, in June 1990 and is currently pursuing a M.S. in computer science at the University of Maryland, College Park, Maryland.

Ms. Newsome has served as chair and vice-chair of the student chapter of the Association for Computing Machinery at Hampton University and is a member of Upsilon Pi Epsilon.

Adopting Ada as a Primary Undergraduate Programming Language

Maj JJ Spegele, USMC and Dr. E.K. Park
Computer Science Department
United States Naval Academy
Annapolis, Maryland 21402

Abstract

Ada usage, as a general programming language in undergraduate institutions, has generally lagged industry demand for Ada-trained programmers and analysts. This paper discusses critical issues associated with adopting Ada as a primary programming language in undergraduate curriculums. Three specific issues, as they relate to Ada, are addressed: Language design and academic programming; compiler requirements for academia; and techniques for teaching programming to undergraduates.

1 Introduction

Commercial acceptance of the Ada programming language has been, and continues to be, impeded by a lack of Ada-trained programmers, analysts, scientists and engineers. For most individuals, initial exposure to programming occurs during their undergraduate educational experience. Universities have traditionally played a critical role in satisfying industry's demand for individuals familiar with commercial programming languages such as C, FORTRAN, Pascal and Assembly. More importantly though, students develop strong opinions about the suitability of particular programming languages during their undergraduate education. Student biases gained during these formative years remain with them long after they graduate. The slow acceptance of Ada in academia has hampered the general acceptance of the language throughout industry.

Today, organization desiring to develop systems in Ada are either forced to absorb the cost of training their own Ada programmers, or to hire previously trained Ada programmers at a substantial premium. For many firms, this additional cost associated with Ada systems development is unacceptably high. Additionally, most organizations are understandably reluctant to adopt an unfamiliar language for any critical or commercial systems development work.

Acceptance of Ada as a widely used, general purpose language is based, in part, on undergraduate institutions exposing students enrolled in technical curriculum to the language. This Ada exposure may range from cursory usage, in an introductory course, to exclusive use throughout a curriculum.

2 Objective

Transitioning to a new language is a non-trivial task for any organization. Our objective is, based on our experiences, to provide insight into the difficulties that may be encountered, and more importantly offer suggestions that will smooth the implementation. While there are any number of concerns and problems that may be associated with using Ada in an academic setting, we see the following as the critical issues:

1. Language design and the nature of academic programming.
2. The availability of tools appropriate for use by inexperienced undergraduate students.
3. Faculty experience and course syllabus.

In this paper, we will address each of these critical issues as they apply to implementing Ada in undergraduate curriculums (both computer and non- computer related disciplines).

3 Critical Issues

3.1 Language Design

In the mid-1970's, the Department of Defense (DoD) identified the need for a state-of-the-art programming language to be used by all the military services in *embedded (or mission critical)* computer applications[1]. Following a competitive design process, Ada was selected as the language that could best satisfy the need for DoD embedded systems.

The primary objective of standardizing on Ada, was to reduce the cost and time to develop DoD software. In general, designers determined this was best achieved by *enforcing* good design and disciplined programming. Specifically, some key design goals were to[2]:

1. enhance program readability.
2. require strong typing.
3. support programming in the large.
4. support exception handling.
5. allow for data abstraction.
6. support concurrency.
7. allow for software reusability.

Ada was subsequently chosen because of its ability to support the development and maintenance of large, complex systems through the application of software engineering principles such as information hiding, abstraction, and specification. It cannot be inferred that all code developed in Ada is inherently well-engineered; rather Ada, when used properly, *supports* the development of well-engineered, maintainable systems.

The language has had unprecedented success in achieving the design goals for which it was intended. Numerous systems in operation, or under development are a testimony to Ada's viability as a language of consequence. Ada is unquestionably a *significant* advancement in the history of programming languages.

3.2 The Academic Programming Environment

An academic programming environment bears little resemblance to the "real world" of applications development. Academic programming is usually done to achieve some specific, short term goal in order to satisfy course requirements. Specific characteristics of the majority of programming done by most undergraduate students in an academic setting are:

1. *Programs are characteristically short.* Program length rarely exceeds 1000 source lines of code (SLOC); for most introductory courses, programs are generally under 100 SLOC.

2. *The project life cycle is compressed.* Student projects rarely extend beyond a few weeks; the majority are completed within a week.

3. *Projects are disposable.* Applications developed are rarely reused - even within a single course. Source code is designed, developed and then routinely disposed of as soon as the project is assigned a grade. There is rarely (if ever) a maintenance phase for code developed in support of academic courses.

4. *Projects are solution oriented.* Academic programming assignments are designed to support the concepts developed in class. In technical courses, the application of this concept to aid understanding is the primary goal. Most scientists and engineers are result oriented; they are not generally concerned with the underlying code. Programming is viewed as a means to solve, or verify, a single problem or concept presented in class.

5. *Projects are isolated applications.* For most course work, students develop solutions in parallel. Rarely do they work in groups that are responsible for individual components of the overall system. In addition, programs developed by students rarely require an interface to external libraries, or routines.

6. *Project design and testing are abbreviated.* Little effort is required, or dedicated to the design of the application. The design phase is often completely overlooked because the student proceeds directly from problem definition (given by the faculty member) to development. Likewise, the testing phase is generally limited because of due-dates and student ambition. Students frequently assume that a successful run, implies testing is complete.

3.3 Ada in an Academic Environment

The preceding discussion highlights the conflict between Ada's design goals and the academic programming environment. Unfortunately, there is little relationship between developing large applications systems and academic projects.

In discussing the use of Ada for academic coursework, we will consider two distinct environments: Ada as a language for computer-related disciplines (computer science, computer engineering, *software* engineering); and Ada as a general purpose language for technical disciplines (engineering, mathematics and other sciences).

Using Ada as a primary language for computer-related disciplines has significant advantages. First, language features directly support traditional computer science topics including data structures (generics); operating systems (tasks), software engineering, and real-time systems. Second, students who are exposed to Ada early in their course of study incorporate fundamental software engineering principles in the design and development of their projects. Finally, the language can, in most cases, be used exclusively throughout the course of a computer-related curriculum. The ability to reinforce understanding and use progressively more advanced features of a single language is invaluable.

Our experience has shown that students, who have some previous exposure to a high level language (C, Pascal, etc.), and who subsequently use Ada over a number of semesters have little difficulty *learning* and *correctly* using Ada. In fact, when provided an opportunity to chose between alternative languages many prefer to work in Ada, particularly when involved in large (by academic standards) projects.

In contrast, teaching Ada, as an introductory course to a general population of students with little or no programming experience, is considered extremely difficult. The fundamental problem is found in the power of Ada. When constrained to the narrow confines of a simple classroom example, it can often inhibit the learning process.

The language is a powerful tool that, in the hands of an expert, produces well-designed, elegant solutions. The language's features however, can overwhelm the average student struggling to produce a 50 line program. For example, rather than appreciating the power of generic packages, the student trying to do simple input/output is left wondering why the language requires instantiating packages in order to write a real number. This adds a level of complexity not appreciated by the student, and in some cases not completely understood by the faculty member.

For those not familiar with Ada, the following example is provided. Most beginning programmers would be able to recognize that the Pascal program Demol will halve a user-supplied integer value.

```
program Demol (input, output);
var
x :integer; y : real;
begin
   writeln('Enter an integer value to be halved.');
   readln(x);
   y := x/2;
   writeln(x,' ',y:0:2);
end.
```

The corresponding code to accomplish the same task in Ada could be written as

```
1  with Text_IO;
2  use Text_IO;
3  procedure Demo is
4  package Integer_InOut is new Integer_IO(Integer);
5  package Float_InOut is new Float_IO(Float);
6    use Integer_InOut, Float_InOut;
7    x : integer := 1;
8    y : float;
9    begin
9        put_line("Enter an integer value to be halved");
10       get(x);
11       y := float(x)/2.0;
12       put (x);
13       put (" ");
14       put_line(y, FORE => 0, AFT => 2, EXP => 0);
15     end Demo;
```

A cursory glance at these two programs highlights the overhead associated with using Ada. Unfortunately, overhead impairs learning - especially for the novice. Experience has shown that explaining the concept of instantiating packages, as shown in lines 4 and 5, to a beginning programmer is a difficult concept. In addition, Ada's strong typing, a significant advantage for software engineers, requires coercing X to type float on line 11. Strong typing then exacts a punishment on the student struggling to understand why anyone would want to type a variable. Finally, Ada's facility for supporting text input and output is a constant source of confusion for projects which are typically I/O intensive (lines 9–14). Unlike Pascal's relatively simplistic (and forgiving) read(ln) and write(ln) statements, Ada's corresponding Put(_line) and Get(_line) procedures are dependant upon the parameter type.

This small example demonstrates how difficult the language can be when applied to simple, introductory problems. Ada's robustness often translates into increased confusion on the part of the student ill-equipped to properly use, let alone understand Ada constructs. Undergraduate programming language faculties are well aware there is a direct relationship between the number of lines of source code and the likelihood a project will fail to compile and/or execute.

For simple student projects, Ada's power, if not masked, can actually inhibit the learning process. The primary purpose of any introductory course should be to teach the fundamentals of problem solving using the basic constructs associated with any programming language (sequence, selection, iteration). This emphasis on problem solving vice syntax will serve the individual - long after the syntactic peculiarities of a specific language are forgotten.

In general, we have found that teaching Ada over a series of courses to students who are strong programmers, is a challenging, but not impossible task. In contrast, those with limited or no programming experience who learn Ada as their introductory language will simply be overwhelmed.

3.4 Compiler Requirements for Academia

Faculty who teach programming languages know all too well that teaching syntax is only half the battle. Correctly using the language by applying it to a problem is the other half. To support problem solving, students require compilation environments that support learning a language.

By their very nature, commercial compiler tools *assume* the user already understands the language - an assumption that cannot be applied to academic programming. Thus, tools, applications, and even languages intended for use in one environment are not necessarily appropriate, or desired in the other. For example, Pascal was originally designed as a teaching language. Pascal's popularity as a language of choice in many universities however, is not reflected in its usage as a commercial applications development language. Similarly, professional programmer's workbenches which significantly improve programmer productivity, provide tools for which in an academic setting would have little or no utility. *Programming tools must support the characteristics of the environment for which they are intended.*

The availability of personal computers in undergraduate institutions has had a profound impact on how students program. This ready access to a personal computer, either in school labs or as student-owned systems, has caused an irreversible change in the methods used to teach programming. Personal computer programming languages and their supporting environments, are generally fast, cheap, easy to use, and yet powerful enough for even the largest student projects. There is a fundamental truth that has evolved: Any programming language used by an academic institution must be adequately supported on personal computers. Evidence of this truth is in the preponderance of teaching materials targeted for any number of popular PC-based programming languages (Pascal, C, C++, BASIC, etc.).

In teaching Ada we have found that a minicomputer-based compiler is not the optimal environment for learning. The language is complex enough without the burden cause by interacting with a hostile editor and an unfamiliar operating system. This is especially true for any introductory Ada course. Experience has shown that a large number of students all compiling and editing relatively small Ada programs on departmental minicomputers can dramatically increase response times to unacceptable levels (10 minutes or more). The result is an unacceptable delay in completing each compile, link and run cycle. This delay and ensuing frustration is a major obstacle to student learning.

One solution is to use personal computer based Ada compilers thereby distributing processor demand over a larger number of smaller processors. A PC Ada environment appropriate for academic use by students and faculty alike, should, as a minimum, have the following characteristics:

1. *Hardware requirements.* The compiler should be capable of operating within 640Kb of memory - with support for using extended memory as either a virtual disk to speed compilation or for compiling larger programs.

2. *User Interface.* The compiler must be surrounded by a easy to use graphic interface. This interface should use the normal combination of menu bar and windows commonly found in most major PC applications.

3. *Language Sensitive Editor.* The editor should support both a normal and a language sensitive mode. Editor commands/keystrokes should be an adaptation of those found in one or more of the most commonly used editors available.

4. *Debugging support.* The environment should support a syntax checker capable of identifying most syntactic errors, prior to compiling. In addition, error messages should incorporate not only the mandatory reference to the Language Reference Manual, but also a supplemental message more appropriate for the novice.

5. *Interface to Libraries.* Any PC-based Ada environment should be supplemented by a variety of libraries

unique to the PC environment. Units must support interfaces to common PC peripheral devices (keyboard, mouse, monitors), low-level operations and finally provide an interface to the operating system.

Validated compilers that meet these basic requirements are becoming available on the market. However, when purchased on an individual basis, the cost is still generally prohibitive for the average student.

3.5 Teaching Techniques

An adequate compiler and environment is a prerequisite, but not a substitute for teaching technique. The best user interface will never overcome a poorly designed course of instruction. Ada's size and complexity makes this particularly important - the language is simply too large to be approached in a haphazard manner.

Teaching a programming language though, should never be accomplished at the expense of the underlying ability to apply a program to solve problems. Too frequently we find students who have "learned" a language but cannot apply the constructs to solve trivial problems. Unfortunately, these students did not learn how to apply a particular language, but rather were taught a language syntax. An overemphasis on syntactic issues obscures what should be gained in any fundamental programming course - fundamental problem solving skills.

Most programming texts, and Ada texts in particular, encourage an emphasis on syntax and style issues. For example, introductory Ada texts frequently address abstract concepts such as packages prior to covering essential language constructs such as selection or iteration. Thus, the student is exposed to the verbiage of the language before knowing how to apply it.

Ada is simply too large and complex to be learned in a single course. To do so only shows the student Ada syntax with minimal opportunity for practical application. Instead, students should be exposed to syntactic constructs and programming structures concurrent with an appropriate level of programming maturity. As an axiom to the principle of Information Hiding, students should only be see the tools they will really need - and nothing more.

To avoid burdening a single course with the full range of Ada, curriculums using Ada should distribute language-specific topics over multiple courses. Figure 1 depicts three levels of Ada teaching: Intermediate, Introductory and Advanced. Each of these levels could easily be accomplished within the contents of a course in a single semester.

Level 1, Introductory, is designed to teach the fundamental concepts found in all high-level programming languages. Topics such as iteration, selection, and the correct use of assignments statements are covered. Note that coverage of Ada types is limited to standard types - for the most part there is no need cover derived types as part of a language introduction. Programs to exercise student understanding should remain very small. Programs that exceed 200 lines in length are unwieldy and counter-productive. Input and output should be limited to covering simple terminal and text file I/O.

The next stage of language learning could be associated with a CS2 or typical Data Structures course. The emphasis is on defining and using abstract data types and applying access type to the manipulations of more advanced data structures such as lists and trees. As they progress, students should be taught how to use and write Ada packages in support of abstract data type operations. Students should also continue to build upon all Level 1 topics, especially in using predefined packages. The Booch components are a typical source[3].

Finally, after significant academic programming experience is achieved, advanced topics such as generics, exceptions, and concurrency could be addressed. These language-specific features are most appropriately covered in an advanced programming course, or as part of a software engineering course. Because of the level of Ada experience achieved at this point (both with the language and the compiler environment) students will be capable of developing much larger and more sophisticated applications. Ideally, faculty would emphasize the importance of completing a fully compiled specification prior to proceeding with the coding phase.

The advantage of this incremental strategy is that a course is no longer a slave to the language. Emphasis shifts from that of "teaching" Ada to using Ada as a tool to reinforce learning. By spreading the learning of the language over multiple courses, the student will more likely develop problem solving skills.

The emphasis on language features at the expense of application is a criticism that applies to most "introductory" Ada texts. As a general rule, these programming texts are often an expansion of the LRM. Unfortunately, many Ada texts are presented in reverse order, covering abstract concepts such as generic packages and derived types before addressing basic language constructs. Faculty must avoid adopting this approach simply for convenience - it will only serve to confuse the student.

An unseen advantage of the incremental approach to learning Ada is that each stage is, in itself, sufficient. That is, the tools provided in Level 1 are adequate to support most quick, no frills, programming tasks associated with an undergraduate engineering project. Again, the student gains experience with only those constructs needed.

Many will argue this defeats the purpose of Ada, as a software engineering language. That somehow restricting the language to its simplest form encourages inelegant solutions that don't exploit Ada unique features. The counter-argument is that Ada provides a suite of tools, from which the user selects depending upon experience and project characteristics. Large, commercial systems with long life-cycles demand elegant, well-engineered solutions; for small-student projects beauty takes a back-seat to functionality.

Levels 2 and 3 then, provide advanced features required by students developing more sophisticated applications in support. Typically, these students would be in a computer-related curriculum. Again, as the concepts and programming tasks become more sophisticated, the students are taught additional tools to support their efforts.

3.6 Ada for the "Masses"

Exposing a general student population to Ada is not viewed as a simple task. Prerequisites for any institution attempting to teach Ada as a primary undergraduate language are:

1. A well-trained and supportive faculty.
2. A well-designed PC-based Ada compiler environment.
3. Selection of texts designed for the novice - not experienced programmers.

	Level 1 Introductory	Level 2 Intermediate	Level 3 Advanced
Topics	standard types - integer - float - character - string - boolean control statements - asssignment - if-then - loop - case input and output - text (file and terminal) arrays - single - multi-dimensional subprograms - functions - procedures - scoping rules packages (using only)	derived types subtypes access types private types unconstrained arrays records packages (writing) binary files - sequential and direct	generics exceptions concurrency
Recommend Project Length (SLOC)	Average — 100 Maximum — 200	Average — 350 Maximum — 700	Average — 750 Maximum — 2000
Taught in	Any introductory programming course (CS1)	Data Structures Course (CS2)	Software Engineering

Figure 1. Three levels of Ada teaching

4. An introductory syllabus limited to those topics identified in Level 1.

In an optimal environment, students will have used a high-level language (preferably C or Pascal), before attempting an introductory Ada course. Students with no previous programming experience will be at a significant disadvantage which may result in unacceptably high failure rates.

4 Summary

This paper has discussed three critical areas associated with the introduction of Ada as an undergraduate programming language. Ada was clearly not designed as an academic language but its ever increasing importance in industry results in a critical shortage of Ada-experienced graduates.

The recent availability of PC compilers appropriate for student use makes possible the wide-spread use of Ada in an academic environment. This language, however, is so extensive that a single course cannot adequately cover its contents. A combination of careful planning and the use of appropriate tools must be used if the adopting Ada is to be succesful.

References

[1] Barnes, J.G.P., Programming in Ada, International Computer Science Series, Addison-Wesley, 1984.

[2] MacLennan, B. J., Principles of Programming Languagues, Holt, Reinhart and Winston, 1987.

[3] Booch, G., Sofware Components with Ada, Benjamin/Cummings Publ Co., 1983.

E. K. Park received his PhD in Computer Science from Northwestern University. His research interests include software engineering and Ada, distributed computing, Ada for parallel processing, and fault tolerance. Dr. Park is currently on the faculty of the Computer Science at the United States Naval Academy.

Maj JJ Spegele is a Marine Corps Data Systems officer presently assigned to the U.S. Naval Academy as a Computer Science Instructor. He has recently taught a variety of undergraduate courses in Computer Science with an emphasis on programming languages and software engineering. His previous billets include tours as an Information Systems Management Officer, Systems Analyst, and Programming Officer at major Marine Corps Data Processing facilities. He is a 1978 graduate of the Naval Academy and holds Masters degrees in Computer Science and Information Systems Management from the Naval Postgraduate School.

CLASSROOM ACTIVITIES FOR AN ADVANCED ADA CLASS

Freeman L. Moore
Texas Instruments Incorporated
Dallas, Texas 75265

Abstract:

An Ada fundamentals course is briefly described to provide the background for an Advanced Ada course. The Advanced Ada course builds upon the knowledge learned in the fundamentals course. While the fundamentals course presents all of the language concepts, sufficient time is not provided for exercising all of the language concepts and features. The Advanced Ada course specifically deals with team programming in its various activities. The activities places an emphasis on reinforcing techniques, not syntax. The course provides an environment for structured experimentation, where information from existing projects is funneled back into the class as part of continual course improvements.

1. Introduction

Texas Instruments has been providing Ada training to its engineers since 1983. We developed our internal training program to be tailored to the needs of the Defense Systems and Electronics Group (DSEG) with Texas Instruments. The mandate from the DoD was that more projects would be required to code in Ada. We responded by developing an Ada fundamentals course and an Advanced Ada course.

1.1 Software Engineering and Ada

Developing an internal training capability requires the integration of several topics. College graduates may be proficient in C or Pascal, but may not have had exposure to Ada, real-time programming, DoD standards and requirements, or much team programming experience. In short, our training program must deal not only with the lack of knowledge about the Ada language, but also the lack of software engineering as applied to DoD software development. Grady Booch's book is a classic, and we strongly support its use to tie together the language and its intended usage.[6] We have developed our own software engineering workshop which introduces employees to DoD-STD-2167A and its applications within DSEG.

Since we are providing training, not education, the challenge is to make the training opportunities as reflective of on-the-job requirements as possible. It is difficult to locate examples which apply uniformly to our audience. This paper will identify some of the programming activities covered in our Advanced Ada course.

1.2 Instructor Competence

We believe the success of any training intervention depends highly upon the caliber of the staff. We have been fortunate to have trainers who have a solid computer science background, and practical on-the-job experience within Texas Instruments. We encourage our trainers and course developers to consult with projects as a means to maintain their credibility.

2. Ada fundamentals class

The Ada fundamentals class will be briefly described in this section. This five day course provides for approximately 25 hours of lecture and 15 hours of hands-on programming activities. The class normally meets every other day over a two week period. The class has a prerequisite of knowledge of a high-level language, although there have been a few attendees with only assembly or Fortran background.

2.1 Hands-on Environment

Attendees receive a "Course completion certificates" upon completion of the course. This requires successful completion of at least 80% of the programming activities. The instructors are responsible for documenting which programming activities have been completed. This is measurable because of our insistence of one-person to one-machine in our classes. The programming labs have detailed instructions for the students. These directions focus attention on getting the task finished in a timely manner, and make sure that learning objectives are met. As time permits, students are encouraged to try their own ideas and examples.

2.2 Change of Hardware Platforms

When the goal is to learn the language and application of various features, the choice of an Ada compiler and supporting environment does not matter much. However, it should be easy to learn, and similar to the job environment. Our course has undergone three changes in machine environments before settling on the current environment, using a PC-AT compatible machine. We are presently using IntegrAda™, but a change is possible soon.

To illustrate the portable nature of the work students have done, a tasking program developed on the PC is uploaded to the VAX™ and executed using DEC-Ada. This provides the opportunity to relate the two programming environments, and demonstrate commonalities between the library environments, and expected capabilities.

3. Advanced Ada class

This section describes the Advanced Ada class and some of its programming activities. The course was first developed in 1985, and has undergone several revisions based upon feedback of prior students, and project experiences about what Ada language features are encouraged and what are discouraged. The Advanced Ada assumes a working knowledge of the Ada language syntax. This class uses that knowledge in a series of activities to reinforce concepts, explore issues, and in general, learn what not to do. The course is four days, generally meeting for two days, off one day, then meeting for two more days.

The course does not use a specific book for a text. References to several book are given in the course notes which students receive. We recommend Cohen, Buhr, Sommerville, and Neilsen as good references.[1, 2, 7, 8]

3.1 Is It Really Advanced?

The course is for people who have already had an introductory Ada course, and want to deepen their knowledge of the language semantics and applications. The key point is 'deepen', since our fundamentals course presents all of the features and syntax of the Ada language. However, within a five day period, there simply is not sufficient time to exercise every language features, nor get involved in discussions regarding style and usage considerations.

OUTLINE

DAY 1	DAY 2	DAY 3	DAY 4
LANGUAGE ISSUES	PACKAGE ISSUES	TASKING TECHNIQUES	LANGUAGE INTERFACING
DESIGN NOTATIONS	GENERIC LIBRARIES	EFFICIENCY / PERFORMANCE	LOW LEVEL FEATURES

Figure 1

3.2 Course Rationale

Since the assumption is made that people already know the language, we have found that this course is necessary since it provides a refresher for those who have not programmed much with Ada lately. The course also a structured opportunity to try ideas in an environment that may not be possible on the job. Lastly, the course is structured to reflect the experiences of various projects with DSEG, and to include that knowledge within the class to benefit future attendees.

3.3 Teams vs. Individuals

The fundamentals course is individual-based to provide the opportunity for everyone to become actively involved in the learning process. The advanced course is team-based. By that, most of the programming exercises involve team effort involving two to four people. For example, members of the team develop separately compiled procedures for a package body. This provides the opportunity to learn working issues as they might apply back on the job.

ACTIVITY: graphical design notations, analyze tasking requirements

A case study is provided which describes a message switching system. The instructor acts as the customer, providing the class the opportunity to ask question and clarify the problem requirements. From the functional requirements, each team determines the top-level CSCI which will become Ada packages, and begins to identify some of the components of each package.

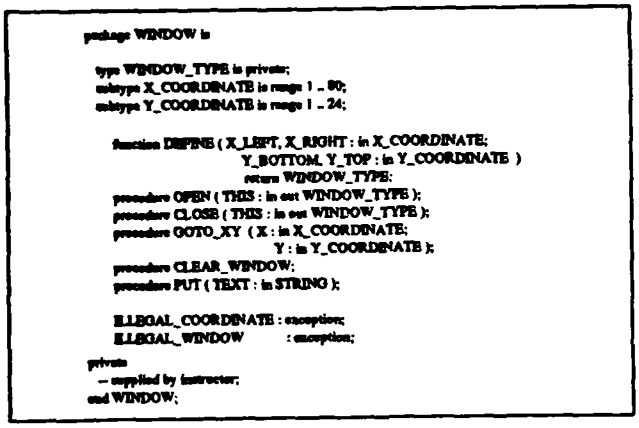
Most of our students have also attended an Object-Oriented Structured Analysis / Design course, and are familiar with the transformation scheme notation of Paul Ward. We encourage the use of graphical design methods, and use the concept of Ada structure graphs as developed by Ray Buhr. Ada structure graphs are used to document the top-level design. Some students will be using Teamwork™ to help in the design process.

Although we don't have access to Teamwork in the classroom, students work together in teams to document the design with paper sketches. These designs are presented to the class as a whole in a short design review. Teams are expected to have sufficiently analyzed the problem to determine the number of tasks required, and the type of rendezvous mechanisms needed. (time = 3 hours)

This case study is revisited in several of the later activities in the course, with smaller activities developing portions of the final implementation. Having a significant case study (10 packages, about 800 lines of code) can be difficult in a compressed amount of time, but the benefits of working on something this size is worth the problems.

ACTIVITY: private data types, separate compilation in a team environment

A simple package specification is provided (figure 2). Using the WINDOW package, teams work together in a programming environment, where each team member is provided with the pseudocode for a procedure. To reinforce the idea of private data types, each team is given with a



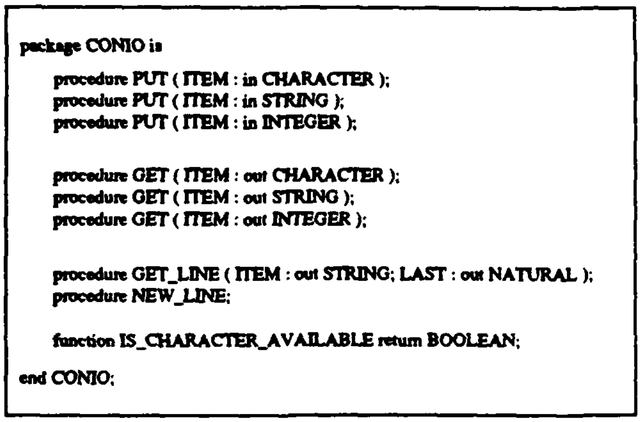
```
package WINDOW is

  type WINDOW_TYPE is private;
  subtype X_COORDINATE is range 1 .. 80;
  subtype Y_COORDINATE is range 1 .. 24;

    function DEFINE ( X_LEFT, X_RIGHT : in X_COORDINATE;
                      Y_BOTTOM, Y_TOP : in Y_COORDINATE )
                      return WINDOW_TYPE;
    procedure OPEN ( THIS : in out WINDOW_TYPE );
    procedure CLOSE ( THIS : in out WINDOW_TYPE );
    procedure GOTO_XY  ( X : in X_COORDINATE;
                         Y : in Y_COORDINATE );
    procedure CLEAR_WINDOW;
    procedure PUT ( TEXT : in STRING );

    ILLEGAL_COORDINATE : exception;
    ILLEGAL_WINDOW     : exception;
private
  -- supplied by instructor;
end WINDOW;
```

Figure 2



```
package CONIO is

    procedure PUT ( ITEM : in CHARACTER );
    procedure PUT ( ITEM : in STRING );
    procedure PUT ( ITEM : in INTEGER );

    procedure GET ( ITEM : out CHARACTER );
    procedure GET ( ITEM : out STRING );
    procedure GET ( ITEM : out INTEGER );

    procedure GET_LINE ( ITEM : out STRING; LAST : out NATURAL );
    procedure NEW_LINE;

    function IS_CHARACTER_AVAILABLE return BOOLEAN;

end CONIO;
```

Figure 3

different private portion of the package, where the instructor defines the data structure. The team must establish any local data structures before individual members write their procedures. After writing the procedures, the members must get back together to integrate their procedures together with the package body, and test it with an instructor main procedure.

A common problem in understanding the problem deals with the coordinates of the window; that is, being consistent with numbering rows and columns of the window and the screen. For people who have just completed the fundamentals class, this simple exercise reinforces the use of private section of a package to hide data structures. They realize this when different teams have different structures to implement, yet the package specification and main program are the same. For the relative newcomers to Ada, working in a team setting establishes the importance of fixing the global data structures defined in the package body before working on any of the 'is separate' procedures. (time = 2 hours)

ACTIVITY: understanding text_io by writing conio

Everyone has used text_io before, and this often generates discussion in the class that text_io is not needed by the various projects because of their embedded real-time nature. Discussion moves to development of test packages, and most people agreeing that some type of I/O is useful, however, not a full-blown text_io package. A package is presented (figure 3) that provides a minimal amount of character input/output. To start the programming lab, the instructor supplies low level test for character, get character, and display character routines. Given these 3 primitive routines, the teams get together and start developing the routines identified in the specification for conio.

This activity reinforces the nature of text_io, in particular realizing that the input of a string variable expects the number of characters equal to the length, not a string ended by carriage return. (Time = 2.5 hours)

ACTIVITY: device control programming, using the serial port

Each computer in our classroom has two serial ports. One port has a mouse connected to it, and the other serial port is connected to the serial port of the adjacent machine. After a discussion of low level features and representation features of Ada, the basics of programming the serial port are presented. Teams of two work together to complete the instructor supplied package serial_io. Teams must develop a main program which accepts characters from the keyboard, and sends them to the other computer via the serial port. Packages conio and serial_io are used. Some students have added the window package to their communications program to have one window show the input, and a second window to show the output. (time = 2 hours)

ACTIVITY: package machine_code and simulators

Because of our job environment, there is a much diversity between processors and compilers needed by various projects. Our training environment is not able to provide specific hands-on training to address all concerns relevant to all compilers. One particularly challenging problem has been trying to develop a programming activity which works with package machine_code.

We have access to the Tartan Ada/1750 compiler and simulators running on a VAX environment. This activity involves completing a short Ada procedure by writing two subprocedures which are completed with package machine_code. After compiling, the programs are run through the simulator and students verify that their assembly instructions were executed. The difficult part of this lab is providing a simplified set of directions for a new compiler and simulator. We admit that this activity does not provide a lot of experience in machine_code. However, reports from students show that the lab is needed to provide a boost to understanding how to interface with other languages.

4. Course and Job relevance

The Defense Systems and Electronics Group of Texas Instruments works with a variety of hardware processors and compilers. Because of resource limitation, we cannot develop specific training for each processor / compiler combination. Our Ada fundamentals course provides training using a PC-based compiler. The advanced course activities described in this paper use a combination of PC-based compiler and VAX-based compiler activities. Handouts are provided on benchmarks of various compilers used within Texas Instruments.

Except for package serial_io, we have refrained from developing MS-DOS specific software in the class. We need to stress the portability nature of Ada, and that initial development can be done in one environment, and then ported to a final test environment.

4.1 Learner Assessment

With the Ada fundamentals course being 40 hours in length, and the advanced course being 32 hours, this represents a considerable amount of time away from the job. We are being asked to document that training is an effective use of time. The fundamentals class includes a pretest and posttest activity to measure leaning, along with requiring completion of key activities. The advanced Ada course does not yet have a posttest, but there is a pretest activity to determine the current capabilities of the students. The instructor is responsible for documenting that all team members work together towards completion of the team's goals.

We have been satisfied with the fundamentals course for the past few years. We anticipate a possible change in compilers soon, along with a more objective-oriented pretest/posttest mechanism. Reports from previous students show that the fundamentals class with its emphasis on learning the object-oriented design and Ada language features and syntax, is well constructed.

4.2 Environment for Experimentation

The advanced course has always stressed team activities instead of individual work. Our direction in developing the course was to provide a structured environment where students can explore ideas and raise questions about efficiency and variations in techniques. Feedback from students indicate that the advanced course packs a lot of activities into four days, all of which are necessary.

5.0 Summary

Texas Instruments has had an Ada curriculum since 1983. Ada training is just one part of the software engineering curriculum which undergoes continual refinement to provide the optimal training for the current job environment. This paper has briefly defined the need and outline of a basic Ada course and an Advanced Ada course. The paper has concentrated on some of the activities used in the Advanced Ada course. The key thrust has been on describing a structured environment where teams work together on different activities to reinforce their knowledge of the Ada programming language.

It has been identified that the cost of the first project done in Ada will be higher than if it had been done with a more conventional language. Providing a structured learning environment where people can practice the ideas and concepts they will need for efficient applications is necessary. This is precisely what we have attempted to do with our Advanced Ada course.

References:

1. Sommerville, Ian and Morrison, Ron, *Software Development with Ada*, Addison-Wesley.

2. Buhr, R., *System Design with Ada*, Prentice-Hall.

3. Wu, Y., "Teaching Software Engineering with Ada", *8th Annual National Conference on Ada Technology*, 1990.

4. Joiner, J., "A First Programming Exercise Using Ada Tasking", *8th Annual National Conference on Ada Technology*, 1990.

5. Schmidth, D., "Ada in an Embedded Real-Time Environment", *8th Annual National Conference on Ada Technology*, 1990.

6. Booch, G., *Software Engineering with Ada*, Benjamin-Cummings.

7. Nielsen, K, & Shumate, K., *Designing Large Real-Time Systems with Ada.*, McGraw-Hill.

8. Cohen, N., *Ada as a Second Language*, McGraw-Hill.

Author's address:

Freeman Moore
Texas Instruments Incorporated
P.O. Box 650311, M/S: 3928
Dallas, Texas 75265

fmoore@skvax1.csc.ti.com

ARCHITECTING DISTRIBUTED REALTIME ADA APPLICATIONS: THE SOFTWARE ARCHITECT'S LIFECYCLE ENVIRONMENT

Walker Royce and David Brown
TRW Systems Integration Group
One Space Park (DH1/2737)
Redondo Beach, CA 90278
(213) 764-3224

Abstract

TRW has completed the development of a reusable *software chipset* known as Universal Network Architecture Services (UNAS) for the Command, Control and Communication (C^3) application domain. To exploit the inherent productivity and quality advantages of standard software components, we have also developed a Computer Aided Design capability known as the Software Architect's Lifecycle Environment (SALE). SALE automates the architectural object bookkeeping and provides interactive feedback to guarantee correct structural syntax and semantics. SALE also encapsulates the UNAS knowledge base so that the logical design for the software architecture can be committed to error free executable Ada source code automatically along with optional models of application performance. This paper describes the SALE approach, in the command and control domain.

Background

TRW has demonstrated impressive productivity and quality advances in the C^3 systems domain through the use of a common layered architecture approa:h, a supporting suite of reusable Ada components called Network Architecture Services (NAS), and automatic source code generation tools which instantiate NAS objects. Using NAS and its support software, the Command Center Processing and Display System - Replacement (CCPDS-R) project achieved approximately a twofold increase in productivity, reliability and flexibility compared to conventional C^3 system development experience. One of the primary contributors to higher productivity was a decrease in the volume of latent software errors inherent in initial test configurations and a reduction in the average resolution time for each error. CCPDS-R Software Problem Report metrics[1] indicate that less than 10% of the development effort was spent doing rework, and less than 15% of the configuration baselines were reworked during development with an average problem resolution time of 15.7 manhours for analysis, implementation and retest. While conventional experience indicates that changes get more expensive with time, CCPDS-R demonstrated that the cost per change with a NAS architecture improved with time. This is consistent with the goals of an evolutionary development approach[2] and the promises of a good layered architecture[3] where the early investment in the foundation components and high risk components pays off in the remainder of the lifecycle with increased ease of change.

We have recently expanded the NAS suite in two dimensions:

1. elimination of NAS' VAX/VMS dependencies through repackaging and redesign into a Universal NAS (UNAS) product which is portable to multiple target platforms, and

2. expansion and integration of UNAS tools into a knowledge based CASE tool called the Software Architect's Lifecycle Environment (SALE).

The previous generation of NAS support software provided powerful architecture development and analysis capabilities. However, it was not general purpose, did not support graphical top level design and provided only a small percentage of the environment support possible. With SALE's expanded graphical design support, tool integration, and automation, architecture baselines of excellent quality are now achievable in manweeks depending on the size and complexity of the architecture.

Figure 1 describes the architecture approach used by CCPDS-R. While this approach has been extremely effective, it could be improved substantially by:

1. eliminating manual tasks

2. providing on-line support for graphical design

3. automating documentation

4. automating smart stub generation so that the architecture provides a continually improving performance evaluation testbed

5. eliminating error prone human translations and communication

6. providing on-line feedback on architecture implications

SALE represents a tangible example of improving the UNAS architectural process in the spirit of Total Quality Management (TQM). Perhaps one of the most important improvements is the elimination of a separate simulation activity for analyzing software/hardware performance. Traditional discrete event simulations for large distributed systems can be difficult to

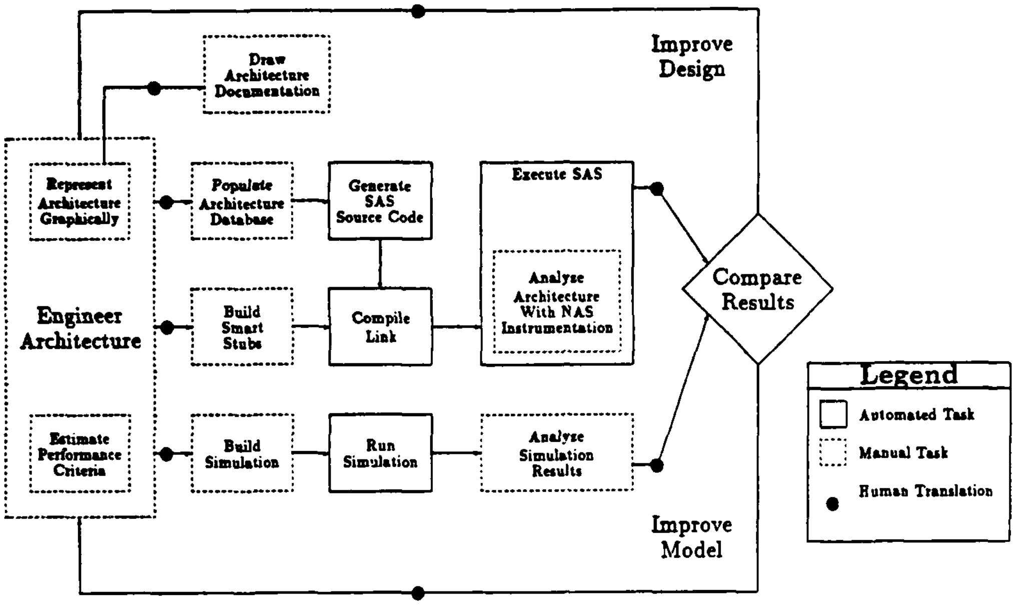


Figure 1: NAS Architectural Process

develop, excessive resource consumers, and typically weak at accurately modeling distributed, non-deterministic functions such as multiprocessing operating systems, multi-tasking Ada runtime libraries and other important components of real-time software performance. With SALE and UNAS, performance analysts need only model true application components. Structural and executive behavior, along with many other computer science oriented functions (e.g., intertask communication, Ada runtime overhead, fault isolation, etc.) can now be *measured* rather than *simulated or modeled.* This not only relieves the modeler of concerning himself with complex, esoteric components, but also increases the fidelity of his results.

Common Architectural Approach TRW's architectural approach, called message based design, has evolved over the last five years from research into successful practice in diverse C^3 applications[3,4]. Message based design is a method for constructing a distributed software solution using a predefined set of architectural objects with well defined operations and functional behavior. In message based software architectures, the top level components are tasks which use message passing as the primary means of data and control flow. NAS provided this predefined set of reusable Ada objects and operations for VAX/VMS based platforms. UNAS provides target independent reusable Ada objects and operations in support of this message based design paradigm.

Software Architecture Skeleton The concept of a software architecture skeleton (SAS) is fundamental to message based design and the incremental development approach prescribed by TRW's Ada Process Model[2]. Although different application domains may define the SAS differently, it should encompass ***the declarative view of the solution which identifies all top level executable components (Ada Main Programs and Tasks), all control interfaces between these components, and all type definitions for data interfaces between these components.*** Although a SAS should compile, it will not necessarily execute without software which provides data stimuli and responses. The purpose of the SAS is to provide the structure/interface baseline environment for integrating evolving components into demonstrations. The definition of the SAS represents the forum for interface evolution between components. In essence, a SAS provides only software potential energy; *a framework to execute and a definition of the stimulus/response communications network.* Software work is only performed when stimuli are provided along with applications components which transform stimuli into responses. If an explicit subset of stimuli and applications components are provided, a system thread can be made visible. The incremental selection of stimuli and applications components constitutes the basis of the build a little, test a little approach of the Ada Process Model[2]. It is important to construct a candidate SAS early, evolve it into a stable baseline, and continue to enhance, augment, and maintain the SAS as the remaining design evolves.

A SAS is not intended to be static. Early stability is unlikely since the information content in the SAS components is likely to evolve with use. To this end, it is important to construct an early SAS which provides a vehicle for interface solidification, and *the flexibility* to adapt the SAS components to accommodate rapid assimilation of design interfaces. It is these SAS qualities which have driven the development and evolution of the UNAS and SALE products.

UNAS Overview

NAS was originally developed from 1984-1987 with the requirement for reusability (a substantial design driver) across arbitrary VAX/VMS networks. It was enhanced on CCPDS-R with added reliability and performance into a product baseline which was reused successfully on multiple VAX/VMS based command and control applications.

An important objective of developing UNAS was to maintain the simplicity of the architectural objects and relationships which need to be understood by the software architect. Figure 2 identifies an abstract view of these objects. An important aspect of the NAS approach was that there were less than 10 types of fundamental objects. These objects can be molded into various forms through a powerful set of tailorable attributes.

UNAS provides components which meet common requirements inherent in all C^3 Systems. By reusing these com-

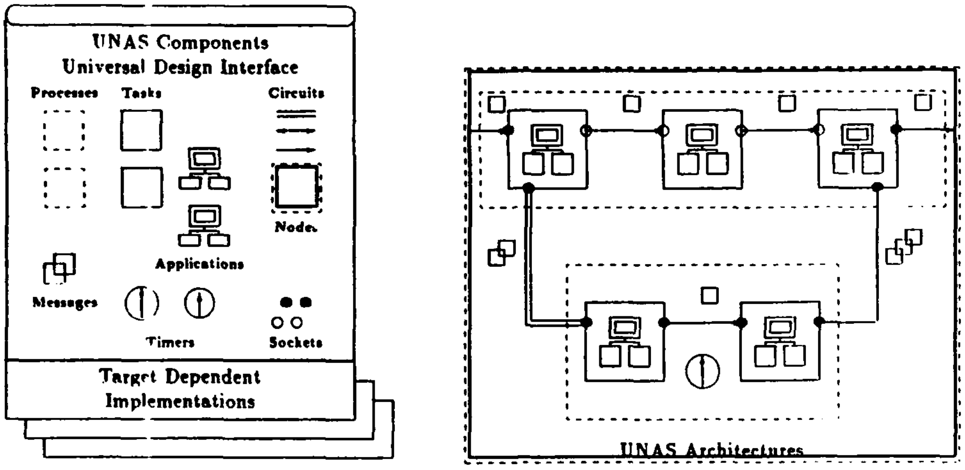


Figure 2: Universal NAS Components

ponents, complex development tasks can be avoided. Furthermore, overall system reliability is enhanced through the use of proven solutions to the reliability critical functions. UNAS provides added product quality and development productivity due to the overall uniformity with which the system is constructed and developed. The "reuse" of design techniques, test techniques, support tools and instrumentation is a substantial benefit whose magnitude is proportional to the size of the application at hand.

With UNAS, an architect is provided a complete "software chipset" for creating logical software architectures. These chips are flexible enough to be "brass-boarded" early in a project's lifecycle to achieve tangible feedback on both structural integrity *(software design)* and operational performance *(software implementation).*

The primary goal in the development of UNAS was minimizing the target dependencies associated with varying implementations of Ada and the interfaces to the target operating system services which UNAS must implement to achieve distributed Ada capabilities. While the Ada language goes a long way towards enforcing standard compiler implementations, there are many instances where a vendor is free to define custom implementations. NAS was constructed specifically using VAX Ada for a high performance application executing on VAX/VMS targets. Consequently, many DEC implementation dependencies crept into the NAS design. Some of these dependencies were incorporated because of convenience, but more frequently, the DEC specific implementation outperformed a more portable solution. The major challenge tackled by UNAS was to repackage (and redesign where necessary) the NAS components to provide the most portable product while maintaining as much of the function, performance and proven design solutions as possible. Another paper[5] describes the UNAS development and retargetting experience.

SALE Description

SALE provides a Computer Aided Software Engineering (CASE) environment to specifically support the software architect's lifecycle needs. UNAS permits the source code for a software architecture skeleton to be produced automatically since it can be represented in a hierarchical data structure and the source code representation is simply a cookbook translation of object names and attributes into predefined Ada program templates and generics. The key objective of SALE is to provide a knowledge based (where UNAS design rules and performance characteristics constitute the knowledge base) environment for architecture design. Similar to the UNAS software chipset hardware analogy, SALE will provide a CAD/CAM environment for UNAS application network design.

Figure 3 illustrates the integrated set of reusable UNAS components and supporting architecture development tools that are collectively integrated into the Software Architect's Lifecycle Environment (SALE).

The SALE capabilities cover the software lifecycle including:

1. A predefined set of network object abstractions (UNAS) for constructing arbitrary Ada task networks. These objects have well defined behavior and multiple parameters for tuning to application specific needs. This software chipset provides for a uniform network construction which is understandable to laymen and expert.

2. Automated source code production for multitask network executives to support rapid prototyping. This capability produces the Ada source code for a network represented in the architecture database, and compiles and installs the network for runtime evaluation. This permits the important applications requirements issues to be tangible early in the lifecycle where solutions can be formulated efficiently.

3. Automatic Task stub production for "tbd" or incomplete components with modeled resource loading. This capability obviates the need for separate simulations of mission performance for architecture evaluation. With SALE, the architecture evolves from an abstract model into a fully functional implementation. The computer science issues (e.g., runtime context switching) which are typically hard to model are visible early in prototyping results. The "abstractions" in the early architecture models are confined to application specific unknowns and continually updated in an integrated performance testbed.

4. An integrated graphical editor for designing, browsing, documenting and specifying the network topography, and defining documenting or reviewing object attributes.

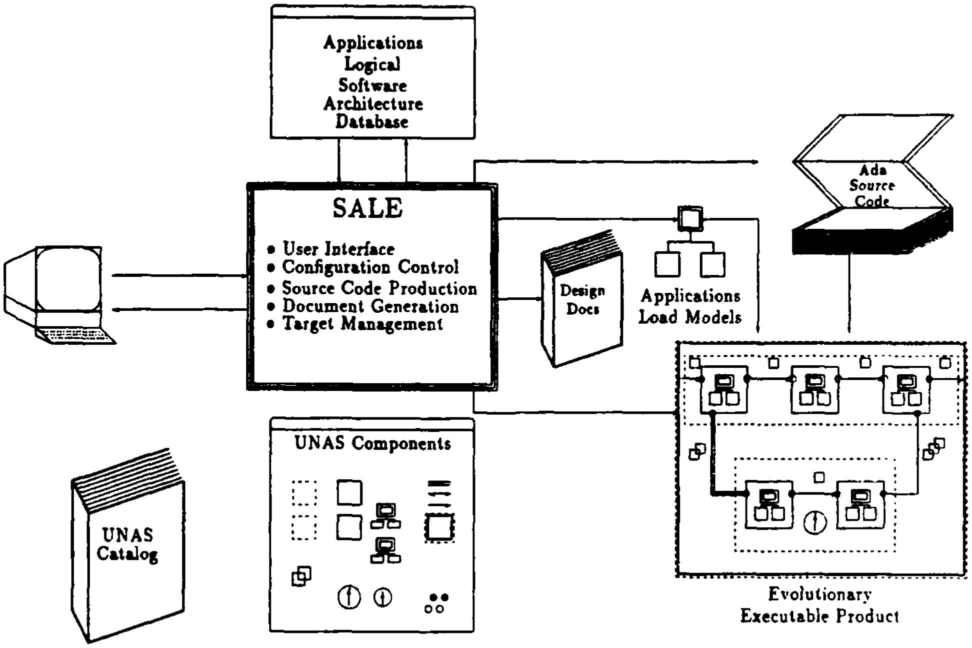


Figure 3: Software Architect's Lifecycle Environment

5. An integrated relational DBMS for accessing, maintaining, evaluating and reporting on the network characteristics.

6. Architecture Documentation production

With the above capabilities, the architectural process of Figure 1 can be substantially improved. Figure 4 identifies the further automation, elimination of error prone human communications, and elimination of separate modeling domains which can be achieved. Although the figure primarily implies productivity improvement, SALE will also result in substantial gains in architectural quality. Much of this improvement is attributable to added flexibility, reduced rework, and ease of change as well as the elimination of error sources. When an architecture is easy to change, a project is more likely to change it more frequently to improve quality.

SALE Foundation The primary capabilities of SALE are layered on top of Systematica Ltd.'s Virtual Software Factory (VSF). In 1990 VSF was evaluated and selected as the basis for building SALE in lieu of developing the capabilities from scratch. In essence, VSF provides a CASE tool for building CASE tools. Creating SALE with VSF amounted to instantiating the "UNAS architectural method" and integrating the supporting source code production tools. VSF provides graphical object manipulation and an integrated DBMS for maintaining design information. These foundation tools were tailored (programmed) to support our UNAS architecture techniques. We installed a UNAS knowledge base into SALE so that architects can efficiently and correctly populate an architecture database. This architecture database is then used as an input to other tools which automatically produce executable models of application programs and a complete executable software architecture skeleton. By integrating these capabilities with UNAS reusable software components and the R1000 configuration management system, a complete SALE for homogeneous networks was developed. This initial operational capability will be extended to support heterogeneous networks in the future.

Current SALE Capabilities During early 1990, an operational SALE was achieved. This effort was accomplished by successfully completing three challenging development tasks:

1. Developing a UNAS graphical object editor. This required the design of the structure of the architecture database and the architect interfaces. The structure of the database had to provide relations for the storage of data and guarantee data consistency at all times. The combination of minimalist set theory and powerful assertion and deletion rules provided by VSF, allowed the design of such a database. When an object is entered (asserted) into the database: it is checked to ensure it has the proper structure; relationships are created to establish default attributes; and more relationships are created to link the object to the rest of the architecture. Rules have also been established to ensure that when an object is deleted, nothing is left dangling in the database. The resulting database structure provides for storage of all objects and attributes required to instantiate a UNAS network, while maintaining a consistent representation of the architecture.

   The interfaces allow the architect to display, enter and modify the structure of the network and the attributes of the network objects. These interfaces take the form of windows in the workstation's native windowing system (e.g., Sunview of DecWindows). The interface definition contains the structure of the window as well as the restrictions for entering data. A different interface was defined for each category of data the architect will enter. When the architect invokes a particular interface, the window is constructed using the interface definition and the current contents of the database. The contents of each window can be captured to a file for inclusion in architecture documentation.

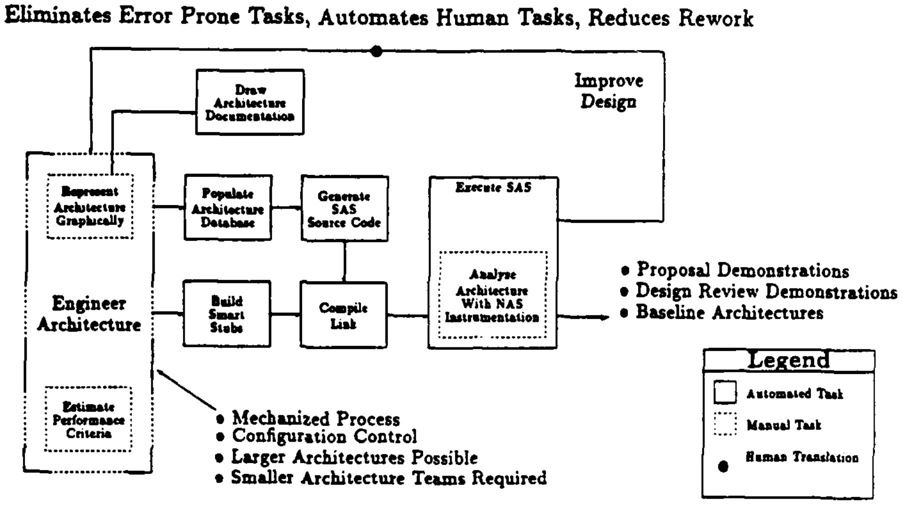

Eliminates Error Prone Tasks, Automates Human Tasks, Reduces Rework
Draw Architecture Documentation
Improve Design
Represent Architecture Graphically
Populate Architecture Database
Generate SAS Source Code
Execute SAS
Analyze Architecture With NAS Instrumentation
Engineer Architecture
Build Smart Stubs
Compile Link
Estimate Performance Criteria
• Proposal Demonstrations
• Design Review Demonstrations
• Baseline Architectures
• Mechanized Process
• Configuration Control
• Larger Architectures Possible
• Smaller Architecture Teams Required
Legend
Automated Task
Manual Task
Human Translation

Figure 4: UNAS Architectural Process with SALE

SALE divides interfaces into two types, graphic and textual. The graphic interfaces are primarily used to represent relationships between network objects. Graphical interfaces capture both the logical and graphical relationships entered by the architect. SALE provides graphical interfaces for entering all network relationships. In fact, an architect can generate all data required for an operational network using only graphical interfaces.

Textual interfaces are structured in a sequential manner. Their structure includes: fields (used to display, modify and enter data); fixed text; and hidden document control characters (used to embed keywords and format control). In addition, structure segments have been nested to allow the structure to be repeated for each database object of a certain type. Textual interfaces provide the following capabilities: automatic generation of architecture documentation; interactive network object attribute modification; automatic Ada source code generation; and interface file generation for use by non-VSF tools.

The combination of the underlying database and the graphical and textual interfaces provide the architect the ability to rapidly define an arbitrary network, modify the network object default attributes, and generate the source Ada code for the network. This code can then be processed (compiled and linked) to generate an executable network. Running the network demonstrates the feasibility of the architecture and UNAS instrumentation provides for measurement of the system performance. The architect can then use SALE to rapidly modify the architecture to incorporate design modifications and change attributes to enhance performance. The Ada code generated includes dumb stubs for the application specific processing. As the real applications are developed, they replace these stubs and the system evolves into the final configuration.

2. Developing a generic application performance model. An important capability of SALE is its ability to generate smart stubs in place of the dumb stubs automatically created as part of the network structure. Each smart stub emulates the performance of an application task. The stubs can read inter-task messages and conditionally expend CPU resources, and write inter-task messages. This on-line entry of performance estimates populates an internal database. The architect enters the same level of detail as would be used to populate a discrete event simulation. This data is then captured in a file and input to a non-VSF tool. That tool processes the file and constructs Ada code which will emulate the application task. This code is completely compatible with the rest of the architecture. After it is compiled and the appropriate processes linked, the network can be run to monitor the performance. The combination of UNAS' ability to measure performance and the performance emulation capabilities of these smart stubs provides an excellent mechanism for performance analysis.

   Conventional simulations require a separate effort to track the software design and construct a standalone simulation to collect performance data. These efforts are constantly trying to keep up with the changing software baseline and often fall months behind. In addition, the accuracy of modeled results is often questioned, both by the customer and the software developers. Our approach eliminates these problems by using the actual software baseline and measuring its performance. The only variable is the accuracy of the estimates used in the smart stubs (which come from the software developers). The accuracy of the measurements increase as the stubs are replaced with the real tasks as they are developed. Thus the same system can be used to obtain performance data from start to finish without the need to expend additional resources.

3. Programming the UNAS knowledge base into the database. In addition to storing data and maintaining architecture consistency, the database has been programmed with rules which enforce compliance with UNAS. These rules ensure that all data contained in the database will result in a properly instantiated UNAS network. They are used to enforce semantic constraints between object attributes. The big advantage here is that correctness is evaluated as the data is entered, giving immediate feedback to the architect. Thus the database always contains a correct design. This is very important

for UNAS since it has been purposely constructed to postpone many semantic checks until runtime. This provides a great deal of added flexibility to UNAS but also increases the penalty of semantic errors. If a semantic error were in the code and not discovered until runtime, a great deal of time would be lost in identifying and correcting the error and then re-compiling and linking the system. SALE's knowledge base removes the possibility of these semantic errors from occurring in the first place. Errors are caught and rejected as the architect enters them.

Future SALE Enhancements While SALE is currently fully functional, there exist a great number of additional capabilities which could be provided. These enhancements primarily represent advances in the amount of automation of the system architect's tasks and improvements in the *ease of use* of SALE. The following items reflect the general future direction of SALE.

- Simplify data movement. VSF provides a Conserve and Merge function for moving data from one database to another. However this capability requires several actions to be accomplished in a particular order. Failure to follow the procedure exactly could result in corruption of the target database. It would be possible to simplify this process to the point where all required data would be captured with a single action and inserted with a similar action.

- Add interfaces. The current set of interfaces is sufficient for constructing a network. Additional interfaces can be added to provide more information to the architect and to produce documentation with different views of the architecture.

- Incorporate modifications for heterogeneous networks. The current version of UNAS supports homogeneous networks. Additions to UNAS to support heterogeneous networks will require additional information in SALE.

- Enhance UNAS knowledge base. SALE currently has a great deal of UNAS knowledge built into both its semantic checks and its help messages. As the product matures this knowledge can be enhanced and broadened to further aid the construction of UNAS architectures.

- Provide a copy capability. VSF does not directly support copying objects. Since this is something which could greatly improve the architect's productivity, we will look at providing an external interface to accomplish this function.

- Automate generation of auxiliary files. Executing a UNAS network requires the existence of several auxiliary files. Automating the generation of default versions of these files would reduce the effort required to setup the system.

- Include non-UNAS objects. Every system will contain objects such as hardware devices which are not related to UNAS. For documentation purposes, the ability to include these objects will be incorporated.

- Provide interfaces to Artificial Intelligence (AI) tools. The information contained in SALE is all that is required to represent the architecture. When this data is combined with performance data collected from the running network, it can be evaluated to optimize performance of the network. This opens the door to the use of AI tools to aid in the architecture design.

Summary

To quantify the benefits of SALE, Table 1 identifies the metrics collected for two different architecture efforts. The UNAS Benchmark is the architecture used to evaluate relative UNAS performance. Forward Area Air Defense (FAAD) is a real-time battlefield system. These initial SALE applications demonstrated the ability to rapidly evolve an architectural concept into a tangibly measurable format in a matter of days. The interactive capabilities and error free implementations permit an architect to concern himself with the important design tradeoffs while relieving the architect from syntactical and semantic implementation details. *Speculation* on runtime performance is replaced by *measurement* thereby accelerating target runtime implementation issues into the design process.

Metric	UNAS Benchmark Value	FAAD Value
No. of Processes	11	8
No. of Tasks	18	8
No. of Sockets	72	33
SLOC Produced	6250	1300
Engineering Time	16 hrs	8 hrs
Implementation Time	8 hrs	4 hrs

Table 1: SALE Metrics

When large, distributed real-time systems can be prototyped rapidly and continuously evolved efficiently, substantial development risk can be avoided. Also, since the software architecture components encapsulate the majority of the top level interfaces, a stable integration framework can be baselined early in a project's lifecycle. This relieves development risk by providing a controlled and unambiguous mechanism for evolving interpersonal communications. The result is a product which maximizes team productivity by eliminating global design issues as early as possible while permitting continuous improvement without fear of significant breakage.

Biographies

David Brown is the product coordinator for the Universal Ada Products for Heterogeneous Distributed Systems Independent Research and Development (IR&D) Project. He is currently responsible for the development and integration of the SALE tool set. Prior to this assignment he was the Subproject Manager for Software Engineering on the CCPDS-R Project from 1987-1990. Previous assignments included roles of increasing responsibility on the Peacekeeper and B-2 programs. He received his BS with majors in Mathematics and Computer Science from Northern Kentucky University in 1980. Mr. Brown has been at TRW for 10 years pursuing the advancement and automation of Software Engineering techniques.

Walker Royce is the principal investigator for the Universal Ada Products for Heterogeneous Distributed Systems IR&D Project. Prior to this assignment he was the Software Chief Engineering responsible for the software development process, the Network Architecture Services software and the overall software engineering on the CCPDS-R Project from 1987-1990. From 1984-1987, he was the Principal investigator of the Software Engineering and Development Division's Ada Applicability for C^3 Systems

IR&D Project. This IR&D project resulted in the foundations for Ada COCOMO (the code cost estimation model), the Ada Process Model and the Network Architecture Services software, technologies which earned TRW's Chairman's Award for Innovation and have since been transitioned from research into practice on real projects. He received his BA in Physics at the University of California, Berkeley in 1977, MS in Computer Information and Control Engineering at the University of Michigan in 1978, and has 3 further years of post-graduate study in Computer Science at UCLA. Mr. Royce has been at TRW for 12 years, dedicating the last six years to advancing Ada technologies for large real time distributed applications.

References

[1] Royce, W.E., "Pragmatic Quality Metrics fo. Evolutionary Development Models", *TRI-Ada Proceedings*, Baltimore, December 1990.

[2] Royce, W.E., "TRW's Ada Process Model For Incremental Development of Large Software Systems", *Proceedings of the 12th International Conference on Software Engineering*, Nice France, March 1990.

[3] Royce, W.E., "Reliable, Reusable Ada Components for Constructing Large, Distributed MultiTask Networks: Network Architecture Services (NAS)", *TRI-Ada Proceedings*, Pittsburgh, October 1989.

[4] Grauling, C.R., "Network Architecture Services: An Environment for Constructing Command, Control and Communications Systems", *Second IEEE Workshop on Future Trends of Distributed Computing Systems Proceedings*, Egypt, October 1990.

[5] Royce, W.E., Blankenship, W.P., Willis, B.P., Rusis, E.A., "Universal Network Architecture Services: A Portability Case Study", *Submitted to Ada IX, 1991.*

Universal Network Architecture Services: A Portability Case Study

W. Royce, P. Blankenship, E. Rusis, and B. Willis
TRW Space and Defense Sector
One Space Park
Redondo Beach, CA 90278

Abstract

TRW has demonstrated twofold productivity and quality gains in the Command, Control and Communication (C^3) systems domain through the use of a common layered architecture approach and a supporting suite of reusable Ada components, tools and instrumentation called Network Architecture Services (NAS). TRW has reused NAS on several C^3 software applications, most notably the Command Center Processing and Display System - Replacement (CCPDS-R) program. CCPDS-R has successfully employed NAS to develop the Common subsystem (350,000 source lines, 10 VAX Nodes, 75 VMS Processes, and 300 Ada Tasks interconnected with 1200 task to task interfaces) on budget and on schedule.

One limitation of NAS however, is that it is dependent on VAX/VMS unique operating system services thereby confining NAS usage to VAX/VMS based target applications. We have recently expanded the NAS suite through repackaging its components into a universal Ada product, Universal NAS (UNAS), with invariant interfaces (user visible specifications) and a minimally variant set of target dependent bodies. The universal parts were engineered on the Rational R1000 to enhance the "pure Ada" integrity of the invariant components thereby ensuring maximal portability. The target dependent parts (approximately 500 source lines) were implemented on various diverse platforms to demonstrate the functional portability and performance difference between retargeted UNAS implementations.

Index Terms: *Distributed Realtime Architectures, Reusable Ada Software, Portability, Fault Tolerant Software.*

Background

NAS/UNAS Genesis. Most C^3 systems have a common core set of requirements (Figure 1) for software executive functions such as initialization, system mode control, reconfiguration, fault detection, health and status monitoring, and interprocess communications. The core components which are constructed to satisfy these common requirements could be reusable between different applications if they were carefully designed and implemented with sufficient power and flexibility. The Network Architecture Services (NAS) product was developed to provide a common set of distributed executive functions for VAX/VMS architectures.

From 1984 through 1987, "TRW's Ada Applicability for C^3 Systems" Independent Research and Development (IRAD) project pioneered the use of Ada and message based design techniques for productive development of large Command and Control Systems. The NAS products were put into practice on the Command Center Processing and Display System-Replacement (CCPDS-R) project with great success. In parallel, other IRAD projects tackled the problems associated with heterogeneous networks and porting message based design products to Unix platforms. In 1990 the Universal Ada Products for Heterogeneous C^3 Systems IRAD merged these lessons learned into a universal approach for message based architectures.

The NAS architecture approach was key to the CCPDS-R success where productivity was double the typical productivity for TRW C^3 software developments. Along with this advantage (and in fact, one of the primary contributors to higher productivity), was a corresponding decrease in the volume of latent software errors inherent in initial test configurations and a reduction in the average resolution time for each error. CCPDS-R Software Problem Report metrics[1] indicate that less than 15% of the configuration baseline was reworked during development with an average SPR resolution time of 2 man days for analysis, resolution and retest. While conventional experience is that changes get more expensive with time, CCPDS-R demonstrated that the cost per change improved with time. This is consistent with the goals of an evolutionary development approach[2] and a good layered architecture[3] where the early investment in the foundation components and high risk components pays off in the remainder of the life cycle with increased ease of change.

The current limitation of NAS is that it only executes on homogeneous VAX/VMS networks. In order to exploit the proven advantages of using NAS techniques, we needed to expand its domain of use into other target environments. This VMS dependency is eliminated by the Universal Network Architecture Services (UNAS) product which provides a single, portable set of interfaces and executes on multiple homogeneous target platforms (CPU, Operating System, Compiler). Ultimately UNAS will be expanded to support a distributed, heterogeneous network so that applications programs can interoperate without knowledge of each other's network residence or platform.

The UNAS Approach

An important objective of developing UNAS was to maintain the simplicity of the architectural objects and relationships which need to be understood by the software architect. Figure 2 identifies an abstract view of this simplicity. An important aspect of the NAS approach was that there were less than 10 types of fundamental objects. These objects can be molded into various forms through a powerful set of tailorable attributes.

The primary goal in the development of UNAS is minimizing the target dependencies associated with varying implementations of Ada and the interfaces to the target operating system services which UNAS must implement to achieve distributed Ada capabilities. While the Ada language goes a long way to-

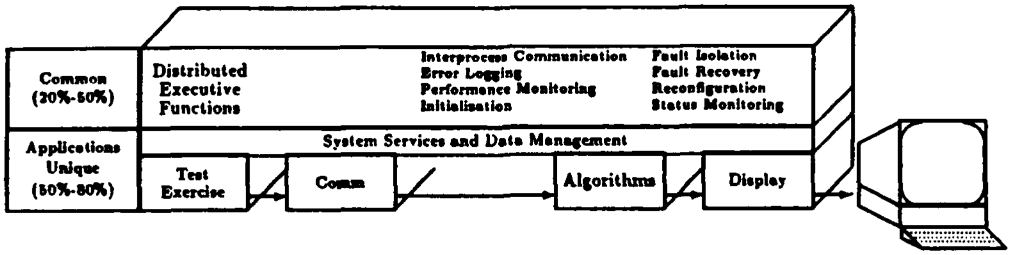


Figure 1: C3 Software Architectures

wards enforcing standard compiler implementations, there are many instances where a vendor is free to define custom implementations. NAS was constructed specifically using VAX Ada for a high performance application executing on VMS targets. Consequently, many DEC implementation dependencies crept into the NAS design. Some of these dependencies were incorporated because of convenience, but more frequently, the DEC specific implementation outperformed a more portable solution. The major challenge tackled by UNAS was to repackage (and redesign where necessary) NAS components to provide the most portable product while maintaining as much of the function, performance and proven design solutions as possible.

The approach for this task is simple conceptually, but very difficult to implement in practice. At the top level, UNAS simply needed to be packaged into an invariant part and a target dependent part. The invariant part should include the complete user visible specifications (to permit applications portability) and the majority of UNAS components. The target dependent part should be confined to a single package body of generic platform dependent functions which are needed by UNAS but not supported directly by the Ada language. Figure 3 identifies the content of the target dependent package body required by UNAS.

Portability features of Ada are easily one of the most esoteric aspects of the language and the quality with which leading Ada compiler vendors support these features vary widely. Furthermore, some functions which could easily be implemented in portable Ada may have potentially serious performance consequences and must also be put in the implementation dependent part to take advantage of higher performance solutions. For example, data movement could easily be implemented using the standard Ada generic function Unchecked_Conversion. However, the performance of this function varies greatly between implementations. In addition, use of this function may be restricted or not even supported in some target environments. By including the means to move data in the implementation dependent part, a portable Unchecked_Conversion instantiation could be used or alternately, an efficient, unrestricted operating system service (such as block move by address) could be used. These kinds of tradeoffs are not obvious without tremendous breadth of Ada experience across multiple vendors as well as substantial experience in the tradeoffs of various NAS/UNAS implementations and their performance sensitivity.

The UNAS redesign of NAS was implemented with two technical efforts. First was the repackaging of NAS to isolate the platform dependent functions into a single package ITC_System_Interface. This package specification provides a generic set of invariant operating system services required by UNAS to implement functions which are outside the scope of the Ada language. The package body then provides target dependent implementations. This separation insulates the remainder of UNAS from being coupled to the underlying implementation and thereby provides complete portability of UNAS and UNAS based applications. The second effort involved the development of pure Ada solutions to a few NAS functions which were dependent on VAX/VMS specific features. In particular, NAS was reliant on VMS logical name capabilities and a DEC layered product called Screen Management Guidelines (SMG). These two functions had to be eliminated in favor of equivalent Ada solutions which would be portable between platforms.

To support these two capabilities the UNAS product includes two new portable Ada components. The DEC SMG interfaces were replaced with portable Ada components which provide the necessary screen/keyboard drivers to allow execution on any ANSI standard terminal device. This removed the NAS dependence on executing on DEC VT220 compatible devices. The logical name translation capability which can vary wide on different operating systems was provided by developing a portable package called NUT_Alias. This package provides the necessary logical name capabilities using only portable Ada input/output mechanisms.

UNAS Experience

The metrics of the UNAS redesign are provided in Table 1. It is interesting to note that UNAS is approximately 5000 SLOC less than NAS (18 KSLOC). This resulted from two phenomena. First, UNAS design incorporated many lessons learned in the design and maintenance of NAS over the last 4 years. This resulted in simpler, more efficient solutions to many of the internal implementations. Secondly, with these new implementations, substantially more internal reuse within UNAS itself was implemented also contributing to the reduction in size. The complete redesign effort required approximately 20 man months spread across an 8 month schedule.

CSC	NAS	UNAS
Generic Applications Control	1200	1000
Error/Performance Monitoring	3100	1300
Interactive Network Management	4200	3200
InterTask Communications	6100	3700
Utilities	1300	1800
User System Interface	3000	2500
ITC_System_Interface	N/A	500
TOTAL	18700	14000

Table 1: UNAS Source Code Metrics

Each of the retarget activities was evaluated by executing a portable applications benchmark. This benchmark configures 16 UNAS objects for evaluating various perspectives of

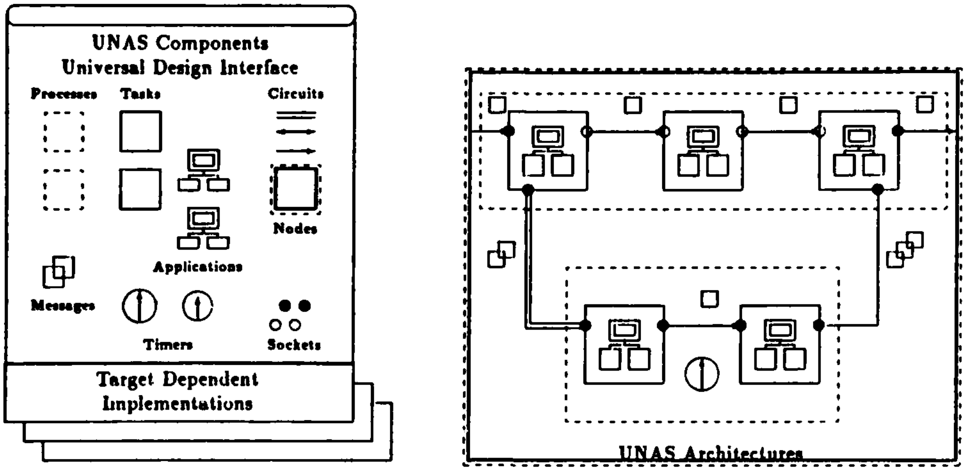


Figure 2: Universal NAS Components

UNAS performance. Three different physical implementations of the benchmark permit the UNAS message passing to be analyzed at all three levels of intertask communication: within a single process (intertask), between two processes on the same node (interprocess), and between two processes on different nodes (internode). Given the various differences in the underlying platforms (processor speed, instruction set architecture, compiler version, etc.), we will not quote the specific results in this paper. It is important to note however, that there were substantial differences in both compilation performance and execution performance as well as the number of underlying compiler/runtime bugs uncovered. Since some of the vendor's compiler platforms are still under non-disclosure agreements, these specifics will not be discussed. Some general differences are noteworthy. Compilation times for the UNAS benchmark (a very Ada generic intensive set of library units) ranged from 30 minutes to 8 hours. Execution times were more similar, all platforms performing within 50% of one another when normalized on a rough "per MIP" basis. Finally, some platforms handled UNAS and its benchmark with no compilation or runtime errors, some required only minor workarounds, and others required sacrificing some of UNAS' portability features (which exploited advanced Ada structures) for less stressful solutions which would be less taxing on the compiler/runtimes but increase the complexity of porting - a standard tradeoff.

The metrics which are quotable for each retarget (Table 2) include the following:

Development Effort Identifies the number of man months required to develop the target dependent part of ITC_System_Interface.

ITC_System_Interface SLOC Identifies the total source lines of code (SLOC) required to implement the target dependent part of ITC_System_Interface.

Implementation Order This number identifies the order in which the retargets were done. The later the order, the more experience the team has in implementing the target dependent part. Comparing these numbers for platform productivity is inappropriate since there are many (intentionally) undefined factors. For example, the team is extremely competent in VMS system services given their NAS experience, but only one team member could really be deemed a UNIX expert.

R1000 UNAS. The Rational R1000 is dedicated to the development and maintenance of Ada software, as well as the configuration management thereof. Being a pure Ada platform (the entire environment is Ada based, including operating system commands implemented as Ada procedure calls), the R1000 is ideal[4] for developing system independent Ada programs. In addition, usage of target dependencies is strictly controlled, making it ideal for cross-development. It is for these reasons that the UNAS development team performs all software development, maintenance, and configuration management on the R1000. The differing implementations of UNAS (i.e., target dependencies) are built and semantically validated on the R1000, then downloaded to and executed on the respective target.

While cross-development is one of Rational's strengths, they also provide a good environment for testing Ada software. Unfortunately, the R1000 does not support any language other than Ada, although files containing "foreign" source code (such as C) can be maintained and managed (but not executed) on the R1000. In addition, it does not support device drives other than those required for a user's terminal. The R1000 is not intended to be a target environment, but a host environment for arbitrary targets. It does however, need to be able to execute UNAS networks to be a credible lifecycle host and support functional testing. The R1000 is therefore, also a UNAS target.

In order to support testing of UNAS based applications and exploit the R1000's advanced functional debugging capabilities, the target dependent portion of UNAS was modified accordingly for the R1000. It was the first target chosen due to the R1000's management of implementation dependencies, resulting in early solidification of this critical aspect of UNAS. The end result was better than 95% portability of UNAS (less than 5% is target dependent). The R1000 environment coupled with broad application and disciplined use of the Ada language allowed this transportability ratio to be achieved. Since the majority of UNAS including the user interface is totally portable, UNAS applications are also portable. Thus, developers may use the R1000 environment for development and testing, and then download and execute their code on the appropriate target without making any source code modifications.

Component	Type
Itc_System_Interface	Package
Byte_Conversions	Generic Package
Convert_To_Bytes	Procedure
Convert_To_Object	Procedure
Deallocate_Byte_Array	Procedure
Information_Services	Package
Get_Cpu	Function
Get_Process_Id	Function
Is_Interactive	Function
Killnet	Procedure
Length_In_Bytes	Function
Move_By_Address	Procedure
Process_Control	Package
Create	Procedure
Delete	Procedure
Failure_Reason	Function
Read_Input_Parameters	Procedure
Serialised_Access	Generic Package
Execute_Critical_Section	Procedure
Semaphore	Task

Component	Type
Process_Io	Package
Close	Procedure
Close_All	Procedure
Failure_Reason	Function
Generate_Name	Function
Is_Open	Function
Name_Of	Function
Notification	Generic Package
Create	Procedure
Notifier	Task
Open	Procedure
Overflow_Routines	Package
Resend_Overflowing_Messages	Procedure
Write	Procedure
Write	Function
Write	Procedure
Write	Procedure

Figure 3: UNAS System Dependent Functions

Once UNAS was completed for the R1000, the UNAS team had an initial baseline from which additional ports could be made, and lessons learned incorporated back into the design. This process will continue as UNAS matures in future versions.

UNIX UNAS. While there was much skepticism concerning the ability of state of the art UNIX platforms to support a state of the art realtime Ada product like UNAS, we found only minor issues in porting UNAS to HP UX (using Alsys) and to Sun SPARC (using Verdix). In general, these implementations performed similarly to the VAX and R1000 versions with some advantages as well as some disadvantages. Overall, the UNAS interface to UNIX platforms is being evolved to remain consistent with existing directions of POSIX.

The major advantage of porting to UNIX systems is that much of the system call interface is consistent from one version of UNIX to another. This allowed us to port over 90% of the implementation dependent part of UNAS from a SUN to an HP platform without changes. This is even more remarkable considering that the UNAS software was not only ported from one UNIX system to another, but also ported to a different Ada environment.

With respect to Unix targets, the target dependent software in UNAS that must be changed is primarily due to differences in Ada environments, not changes in UNIX systems. Nowhere is this worse than in interrupt (UNIX signal) processing. Although each vendor we have used does enable UNIX signals to be tied to Ada tasks, each interface is different. Some vendors give full interrupt functionality, while others may provide a subset. Still others have changed the interrupt processing such that it is unique to their Ada system.

The best way to alleviate this problem is to standardize the Ada interface to UNIX. This is currently being done in two steps. The first is adoption of the POSIX interface, which alone should reduce many of the problems associated with porting programs from one UNIX platform to another. This, however, is just an interface to a UNIX-like operating system, based on the C language. In many instances Ada vendors must make decisions on how to support a particular feature (such as signal handling). Naturally, the decisions, and hence the implementations and interfaces, will differ from one vendor to the next.

In response to the wide range of Ada interfaces to UNIX, a section of the IEEE is working on an interface for the Ada language to POSIX. The interface definition is not simply Ada bindings to C. Rather it is a comprehensive effort to facilitate the use of all POSIX features in a way that is consistent with the Ada language. For example, the handling of interrupts in Ada can be tied to task entry points in a fairly portable manner.

Once the Ada interface is finalized, and supported by major Ada vendors, the task of porting programs from one Ada vendor to another should be much simpler and hence more cost effective. Such a standard would be quite welcome to the developers of UNAS, as well (we are sure) to other projects that attempt to program in Ada for UNIX based platforms.

HPUX/R1000 CDF. As mentioned previously, the Rational R1000 environment provides an excellent cross-development capability. Although target dependent Ada code may be developed and semantically validated on the R1000, it must be downloaded to the target, and then compiled, linked, executed, and debugged using some other vendor's compiler. To alleviate this cumbersome process, Rational provides a tool known as a "Cross-Development Facility" (CDF) for some specific targets (we specifically used the M68000 version for HPUX). The CDF not only provides the standard R1000 capabilities for target specific software development and configuration management, but also provides the capability to compile, link, execute, and debug on the target from the context of the R1000 environment. Using the CDF, one need never login to the target upon which the program is executing.

The CDF fully automates the host-to-target software lifecycle. In addition, the CDF utilizes a Rational provided Ada run-time, rather than making it the developer's responsibility to acquire and learn some other vendor's compiler for use on the target. This assures compatibility between uses of implementation specific Ada features and the particular run-time, in that they are both provided by Rational. The CDF also saves time and effort in that after compiling on the R1000, the code need only be linked and executed from the R1000. The CDF automatically downloads code to the target as needed, transparent to the developer. Without the CDF, the code would require manually downloading and recompiling on the target before linking and executing.

From the developer's perspective, the fact that their program is running on another machine is almost entirely transparent. All I/O performed by the program is done in the context of an R1000 window, just as if it were running on the R1000 and not the target. The CDF makes the R1000 a fully functional "Universal Host" for certain target environments, with full run-time support for mixed language programming

Criteria	R1000	VAX VMS	Sun Unix	HPUX	HPUX CDF (R1000)
Development Effort (MM)	2	.25	2.5	2	2
ITC_System_Interface SLOC	488	570	752	719	690
Implementation Order	1	2	3	4	5

Table 2: UNAS Retarget Implementation Metrics

and hardware interfacing. In addition, developers can use the advanced R1000 debugger to debug programs executing on the target.

VMS UNAS vs VMS NAS. VAX Ada provides an adequate development system as well as a highly sophisticated run-time environment. Mixed-language programming, VMS System Services, VMS Run-Time Library Routines, access to VAX Macro machine instructions and other aspects of VMS are easily accessible and fully supported by VAX Ada. VAX Ada also supports system independent Ada programs, with compiler generated source code listings including a portability summary with information on system dependencies present in a program. Thus, a programmer may choose portability and determine via compiling whether or not their code is truly portable. Or, a programmer may choose to take advantage of VAX Ada and VMS specifics to write complex device drivers and the like.

Having had many years experience with VAX Ada, and since VAX/VMS was the original NAS platform, porting UNAS to VAX/VMS was relatively simple. It took approximately 1 man-week to modify the system dependent portion of UNAS accordingly, and to fully debug and test it. Although the UNAS internals have changed greatly from NAS, familiarity with VMS System Services in general and UNAS' design for portability allowed us to quickly release the VMS version of UNAS. Other factors directly influencing the speed of the port were VAX Ada's excellent run-time support, and the maturity of VAX Ada as a virtually "bugfree" compiler.

Runtime performance of the portable UNAS was expected to be slightly worse than the VMS dependent NAS since, in general, portable solutions tend to have more inherent overhead. However, UNAS also incorporated many design lessons learned in its internal structures and operations which offset the portability overhead to achieve even higher performance on VMS than NAS could achieve. Specifically, the initialization time performance (substantially improved) and task to task message throughput (20% improvements) are attributable to innovative design solutions which incorporated specific knowledge of Ada runtime performance issues.

Portability Lessons Learned

UNAS portability implies much more than just having UNAS components written in Ada that will compile on any platform. UNAS provides multiprocess control, networking, interprocess communications, CPU performance monitoring, and interfaces to the user defined UNAS program configurations. All of these objects are outside the domain of the Ada language standard. This means that UNAS is not only portable with respect to Ada, but also creates Ada compatible abstractions for the isolated operating system, platform, and network protocol interfaces.

While writing portable Ada software that is, in and of itself, heavily system dependent *is not easy in Ada*, in any other language it would have been even more difficult. Ada language standardization and the ability to abstract system dependent data objects and interfaces made the development process manageable. Even so, there were many instances where design changes were necessary in order to make it more general. For example, UNIX named pipes result in message fragmentation when writing messages greater than a certain size. Conversely, several messages may be received in a single I/O operation at the destination. Consequently, the UNAS internals had to be modified to allow for message "packetization" at the source, and message "assembly" and boundary determination at the destination. While these modifications were made for UNIX platforms, they also greater generalized the UNAS internals to handle arbitrary message sending and receiving patterns, as opposed to assuming a "one message written/one message read" scenario. Thus, each UNAS platform provides additional lessons learned that, once incorporated, greater generalize the design.

A single solution that will work in all implementations is highly desirable, and while it takes a great deal of effort to achieve it, portability is possible. However, one must take care that the pure solution does not hinder performance significantly. While UNAS is currently operational on the target environments discussed above, performance of interprocess communications could stand improvement on the UNIX platforms. Portability was traded off for performance in order to simplify the retargeting process, so that UNAS could be operational on a variety of targets quickly. Rapid retargeting resulted in substantial lessons learned early in the design, and also proved that UNAS technology works on arbitrary targets. Having done this, the UNAS internals can now be modified to take better advantage of implementation defined features, thus improving performance, at the expense of portability. Nonetheless, it is anticipated that UNAS will still be 90% portable after these modifications.

Since the UNAS interfaces were solidified long ago, UNAS applications need only be recompiled against new versions, and will not require modification. This is another Ada advantage: abstraction of UNAS objects for the application and isolation of the UNAS interfaces into a standarized package specification. UNAS will continue to improve with time, both performance-wise and target supportwise, independently of UNAS applications.

While porting UNAS to various targets, several lessons were learned and two distinct classes of portability were observed:

1. "Internal" Ada portability. Some of the compiler implementation dependencies were immediately obvious and were pointed out by some of the compiler reference manuals. Use of implementation defined packages (e.g. System, Low_Level_Io, Interrupt, Os_Files, Starlet , etc.) was avoided or isolated to UNAS system dependent package bodies. Use of implementation defined pragmas or attributes was similarly limited. While unrecognized pragmas should be acceptable (with possible warnings) or ignored by any foreign Ada compiler, care was taken so that a pragma used by several compilers would not result in different meanings (e.g. pragma Main).

More subtle implementation dependencies (in package Standard) were also avoided. Dependence on implementation defined range constraints of Integer, Duration, etc. was avoided by use of explicit "subtypes" constraining the ranges to the minimum required by UNAS.

Unconstrained or variant objects were not used in inter-process communications of UNAS defined messages, because of various implementations and restrictions on use of Unchecked_Conversion, Address, and Size attributes when converting these messages to internal byte stream formats.

Use of record or array aggregates and "others" clauses generally included the explicit type of the object, to avoid pushing the compilers "to the edge" into areas subject to interpretation of the Ada language standard.

UNAS Ada task specifications were carefully designed to support implementation dependent task priorities, task stack size control, and acceptance of system interrupts at rendezvous entry points. Task bodies were designed not to rely on task scheduling, priority, or pragma Shared for synchronized resource control, or access to minimized shared data objects.

2. "External" environment interface portability. In order to perform operating system dependent inter-process communication, UNAS Ada abstractions were created, in an invariant package specification, for "pipe-like" objects used by UNAS. These abstract UNAS devices were then implemented in the system dependent package body and utilize VMS mailboxes, UNIX pipes, or shared memory, etc., on each respective system. This allowed for a very loosely coupled implementation, and the majority of UNAS code to remain invariant across the various targets.

Remote process creation was similarly implemented. It was assumed that any operating system process model requires Input, Processing, and Output to be specified. An invariant Ada procedure specification was designed, that required three string parameters specifying the standard input, standard output, and executable file names. The body of this procedure was then free to make any operating system calls necessary to implement the function.

Along with the need for portability, needs for relocatability and run-time setup parameters emerged. In order to implement an efficient run-time parameter passing mechanism (to a process) in a portable fashion, UNAS Alias Names were developed. UNAS Alias Names are string literals that are assigned string values, which can be accessed at run-time by Ada procedures (UNAS Alias Names are equivalent in concept to VMS Logical Names or UNIX Environment Variables). UNAS Alias Names are implemented using the implementation-independent Ada standard package Direct _Io. The Ada interface to UNAS Alias Names is provided by a UNAS provided package NUT_Alias. This package is used extensively in UNAS and is accessible by UNAS users. It allows modification or redirection of UNAS setup parameters, system configurations, file names, directory names, process names, node names, timeout parameters, etc. without recompiling any Ada programs, and in a similar manner on any platform. These services allow avoidance of "hard-coding" any explicit filenames or directory names which are operating system dependent. Use of Alias Names is optional for a UNAS user, and does not preclude or obscure visibility to any operating system dependent logical names, aliases, or environment variables, which are often used to implement hierarchical development testbeds.

The generic Ada UNAS User Interface, used to implement the Interactive Network Management operator console displays, was re-designed to eliminate dependencies on the VAX/VMS Screen Management Guidelines (SMG). A conceptually similar screen driver was developed in Ada based on ANSI escape sequences output by Text_Io.Put_Line. This has allowed even the device dependent portion of the screen management procedures to be portable on most of our platforms, with terminals ranging from serial VT100 devices to X-Windows, since they all support ANSI. However, the keyboard input portion of the User Interface requires device dependent drivers for each system, because of the various methods employed by operating systems for blocking processes while waiting for input, whether it uses Text_Io.Get or operating system calls. Also, function key escape sequence validation and "no echo" characteristics are managed differently by each system. To overcome the variations in function key definitions on different terminals, the User System Interface is equipped with a keystroke recording and mapping procedure automatically invoked when first installed. This allows the system unique character sequences to be learned and remembered in a "table look up" fashion, thus making the character sequence validation and look up routines entirely portable.

MultiTarget Configuration Management. Configuration management of UNAS source code, under development for the multiple targets, was performed on the Rational R1000 using the Configuration Management and Version Control (CMVC) system. This system was chosen because it was the only configuration management system that met our needs, and for managing Ada software, it provides far more capability than any other known CM system. For managing UNAS configurations, CMVC provides us with the following capabilities:

1. Management of Ada units - The CMVC system has a head start on all other CM systems, since it deals directly with Ada units, not with system dependent disk file names, to identify a controlled unit of code. This immediately relieves the burden of cross-referencing the same Ada unit on different targets, using different names, and allows the CMVC system to interact directly with the required Ada library environment. Other CM systems are independent of the Ada Program Support Environment (APSE), and rely on disk file names.

   CMVC is based on Rational's "subsystem" concept, which enforces logical groupings of Ada packages, restricts visibility to any system-dependent packages, and reduces the common Ada "withing" wars that result on larger projects, if scope of visibility and "withing" order is not controlled.

2. Allows multiple compiler models - CMVC allows the installation of any other Ada compiler's environment interface packages, thus creating an equivalent compilation environment, in which we can semantically validate and analyze code for a foreign target system. By using CMVC, all of the UNAS code for the various targets can reside (in a pre-compiled state) on one machine, where it can be easily controlled.

3. Executed on development system - CMVC resides on the same system we perform all of our development. This way it controls software in development, and elimates any machine-to-machine transfer time (to return it to configuration control). Also having an executable version of UNAS for the Rational R1000 allowed much of the unit testing to be performed there, using controlled configurations.

4. Unit reservation - CMVC supports reservation of Ada units for modification. This greatly reduces the chances of losing simultaneous edits to the same unit by different individuals.

5. Multiple version and release control - CMVC allows multiple releases of software to coexist on the same system. This way, one developer can make changes and test them, concurrently with another developer. Stable versions can be "frozen." The CMVC subsystems also

allow Ada specification to be modified in one subsystem, and not immediately impact dependent units in other subsystems.

6. Change tracking and recovery - CMVC internally maintains the differences between all generations of a controlled Ada unit. This allows reconstruction of any previous generation, or examination of line-by-line changes.

7. Change propagation into common invariant units - CMVC allows UNAS device independent units, which are to remain identical across targets, to be controlled as one configuration object. If a developer changes a "joined" unit in one target implementation, the same changes are automatically enforced on the other target versions of the same unit. The automatic propogation of these changes can be postponed (but not avoided), if a specific target is not to be impacted at the time of the change.

8. Incremental modification - Modification of controlled Ada specifications can be done incrementally, i.e., if the change does not directly impact a dependent unit (e.g. adding something new to the specification), the Rational Ada environment will not require that the dependent unit be recompiled.

The weaknesses of the Rational CMVC system (as well as most other Ada CM systems) are that 1) it is very disk intensive and 2) it cannot handle implementation defined attributes, such as the VAX/VMS Ada specific " 'Ast_Entry." Currently, device dependent attributes are preserved by changing the line to a comment, with a special marker, that must be removed when the code is downloaded to the actual target. (This type of line count is less than 10.)

Summary

Positive experiences in reuse and portability are few and far between. The NAS transition to UNAS represented a complex challenge in portability. Since the inherent purpose of these products was to isolate applications from the underlying implementation complexities of the host/target virtual machine, this effort was focusing explicitly on the most target dependent parts of Ada and the boundaries of the language definition. The standardization of the Ada language as well as disciplined, careful design resulted in a substantial success story in portability.

The UNAS product has been retargeted to VAX/VMS, SUN Unix, HP Unix, and the R1000 target platforms. Future retargets to IBM's AIX, MAC II, VAX Ultrix and PC DOS are planned. This paper has described our experience in redesigning and repackaging UNAS as a case study in Ada portability and our lessons learned in retargeting to multiple platforms. In summary, we believe the number one lesson learned is that *building reusable and portable components is extremely difficult*. It takes highly experienced personnel, sufficient schedule and excellent tools. But even more importantly, it takes real world users and usage on real world projects[3,5,6] before the adjectives **reusable** and **portable** can be used legitimately to describe a product.

Ada Cocomo[7] quantifies the complexity of building reusable components in a range from 0% to 50% added effort. After many man-years of reusable and portable component development, this range appears appropriate. We estimated that NAS (reusable, but not portable) required 30% more effort so that it could be reused in three CCPDS-R subsystems without modification. UNAS clearly required at least 50% more effort (and undoubtedly 50% more schedule) to also achieve portability. The complexity of portability ensures at least a two pass design effort and necessitates a small cohesive team. Ignoring these circumstances in future reusable/portable development efforts will result in substantial risk of failure.

On the positive side, UNAS provides a tremendous layer of isolation with which applications components can now be implemented in a reusable fashion. The investment in UNAS will not only pay off in near term usage by applications projects but also in providing leverage to achieve even higher levels of reuse in future projects. To exploit the inherent productivity and quality advantages of standard UNAS components, we have also developed a Computer Aided Design capability known as the Software Architect's Lifecycle Environment (SALE)[8]. SALE automates the architectural object bookkeeping and provides interactive feedback to guarantee correct structural syntax and semantics. SALE also encapsulates the UNAS knowledge base so that the logical design for the software architecture can be committed to error free executable Ada source code automatically along with optional models of application performance.

REFERENCES

[1] Royce, W.E., "Pragmatic Quality Metrics For Evolutionary Development Models", *TRI-Ada Proceedings*, Baltimore, December 1990.

[2] Royce, W.E., "TRW's Ada Process Model for Incremental Development of Large Software Systems", *Proceedings of the 12th International Conference on Software Engineering*, Nice France, March 1990.

[3] Royce, W.E., "Reliable, Reusable Ada Components For Constructing Large, Distributed Multi-Task Networks: Network Architecture Services (NAS)", *TRI-Ada Proceedings*, Pittsburgh, October 1989.

[4] Royce, W.F., Development of Reusable Ada Packages Using The VAX 8600 and the Rational R1000 Ada Environments. Proceedings of "Methodologies and Tools for Real-Time Systems" Conference, National Institute for Software Quality and Productivity", September 8, 1986.

[5] Grauling, C.R., "Network Architecture Services: An Environment for Constructing Command, Control and Communications Systems", *Second IEEE Workshop on Future Trends of Distributed Computing Systems Proceedings*, Cairo Egypt, October 1990.

[6] Grauling, C.R., "Pilot Command Center Testbed Development Environment: A better Way To Develop C^3 Systems", *Submitted to Ada IX, 1991.*

[7] Boehm, B.W., Royce, W.E., "TRW IOC Ada COCOMO: Definition and Refinements", *Proceedings of the 4th COCOMO Users Group*, Pittsburgh, November 1988.

[8] Royce, W.E., Brown, D.B., "Architecting Distributed Realtime Ada Applications: The Software Architect's Lifecycle Environment", *Submitted to Ada IX, 1991.*

CATALYST
AN INTEGRATED SOFTWARE ENGINEERING ENVIRONMENT

Sandra L. Mulholland

Rockwell International, Collins Commercial Avionics
Advanced Technology and Engineering Division, Cedar Rapids, IA

Abstract

Within the Rockwell International, Collins Commercial Avionics, Advanced Technology and Engineering Division (AT&E), there is an on-going effort to develop an Integrated Software Engineering Environment (ISEE) that supports both government and commercial projects. The name of this ISEE is "Catalyst." This paper focuses on some of the challenges presented in ISEE development: selecting an ISEE standard on which to base environment development; establishing an evolutionary environment development path; and defining those functional capabilities that support a full software development life cycle. A brief discussion of the benefits of having the ISEE used by real projects at the same time as it is being developed is also provided.

Background

To be competitive in today's complex software market, corporations must build systems exhibiting high quality, cost efficiency, and compliance with increasingly, sophisticated government and commercial development standards. In order to achieve systems which meet these criteria, corporations must: 1) apply software development standards in a prudent and consistent manner; 2) apply a consistent, uniform process to all software development; 3) automate the development process where possible; and 4) utilize an integrated software engineering environment (ISEE) which supports these requirements.

In August, 1985, an environment development plan[4] was written which identified the software development needs and goals within the avionics organizations of Rockwell. This plan was used as the basis for an environment development project which was started in October, 1987, by the Avionics Advanced Technology and Engineering (AT&E) organization.

In 1989, a Software Engineering Environment (SEE) Section was formed within AT&E that took over the environment development project and expanded its scope. The SEE section charter includes developing an ISEE which :

- Supports a high quality, software engineering process;
- Supports a full software development life cycle;
- Supports multiple software development standards *(commercial and government)*;
- Supports multiple development languages;
- Supports systems of varying scopes and sizes *(e.g., rapid prototype vs. full development, 3000 lines of code vs. 100,000 lines of code, etc.)*;
- Supports multiple application domains *(e.g., real-time, embedded avionics; data intensive, real-time; display management applications, etc.)*;
- Supports and encourages software artifact reuse;
- Automates formal and informal document generation / verification;
- Is available on a distributed, heterogeneous platform;
- Complies with, and embodies, applicable ISEE standards;
- Supports an evolutionary environment growth path providing an effective, useable ISEE at all stages of environment development;
- Maximizes the use of commercial off-the-shelf (COTS) products; and

- Exhibits the characteristics of tailorability, flexibility, extensibility and enhanceability.

"Catalyst" is the name which has been given to the ISEE developed under this effort. The name is not an acronym, but simply identifies the significant, positive change to software development that occurs as a result of the use of the ISEE.

Catalyst

In December 1990, Catalyst, version 1, was completed. In this version, the major goal was to develop an environment that supports an engineering process which would :

- Result in the automatic generation of DOD-STD-2167A[3] data item deliverables (DIDs) for the Software Requirements Analysis and Preliminary Design life cycle activities;
- Establish good basic management capabilities such that systems developed utilizing Catalyst would enjoy a uniform level of high quality; and
- Encourage the reuse of project artifacts.

Challenges.

In order to accomplish this major undertaking, the following challenges were presented:

- Predict what emerging ISEE standard(s) will apply in the future and ensure that Catalyst will be compliant;
- Develop Catalyst in a manner that provides constant SEE support to on-going Rockwell projects at the same time that the environment itself is being evolved to support a full ISEE concept;
- Define those functional capabilities that must be provided by an ISEE in order to fully support the software development life cycle.

In addition to these challenges, the resulting environment had to:

- Support multiple development languages; even though the Cedar Rapids facility of Rockwell is a very significant Ada developer, work is also being performed in other languages, such as C ;
- Be available on heterogeneous, distributed platforms with a common user interface;

and, what is always a critical challenge,

- Be successfully inserted into corporate production efforts.

Applicable ISEE Standards.

Because an ISEE comprises many interacting components, there is an abundance of standards efforts and/or definition efforts, which may apply to an ISEE itself, or to individual ISEE components. The fast pace at which technology is advancing in this area has created a situation where, in some cases, more than one effort is targeting the same (or like) ISEE component for standard/definition development.

For example, MIL-STD-1838A, PCTE+, ATIS (CIS), and several other efforts target tool integration; DIANA and IRIS target data structure representation; GKS and PHIGS target graphical data representation; and MOTIF and Open Look target a common window presentation definition.

In other cases, the objects involved in one standard/definition effort are the same objects that are used in another effort, and yet the object definitions are different. For example, the IEEE P-1175 "A Standard Reference Model for Computing System Tool Interconnections"[1] and the Department of Defense (DOD) Computer Aided Acquisition and Logistics Support (CALS) efforts have each defined classes, attributes and relationships for some of the same objects. Because there was no initial coordination between the two efforts, at the present time inconsistencies exist between the two definitions. In other words, it is feasible that in the future an ISEE may be tasked with supporting projects that have requirements for P-1175 as well as projects that have requirements for CALS. In this situation, the objects created in a CALS-governed project will not be understood by objects created in a P-1175-governed project. In order to create data interoperability between the two projects, a filter or "translator" would have to be built to convert the objects created under the P-1175 definition to the CALS definition and vice versa. This situation compounds the massive translation problem that already faces environment builders. It also makes it almost impossible for environment builders to easily determine which standard/definition best serves their purposes or which one *should* be applied as it is the standard most likely to be required on government or commercial contracts.

Recent events show that the difficulties of this situation have begun to be noticed. For instance, in a letter dated October 25, 1990, Robert M. Poston, Chairman of the IEEE Computer Society Task Force on Professional Computing Tools, announced the intention to coordinate the IEEE P-1175 definition with that of CALS, IRDS, PDES and CDIF. Another significant coordination effort that has been undertaken is the "merging" and redesign of the PCTE$^+$ and MIL-STD-1838A definitions into the Portable Common Interface Set (PCIS)[2]. This coordination between complementary or competing standard/definition efforts must be achieved in order to establish a firm path for environment developers.

Even with all of the present confusion, for future extensibility and standards compliance reasons, environment builders must base their development efforts on an emerging ISEE standard. Two of these emerging standards are the National Institute of Standards and Technology (NIST) ISEE Reference Model and the European Computer Manufacturers Association (ECMA) Task Group TC33/TGRM Reference Model[6]. Another effort which may be considered to have major influence in the establishment of ISEE standards, is the Software Technology for Adaptable, Reliable Systems (STARS) program. The following paragraphs provide a brief discussion of each of these efforts and a justification for the ISEE standard selected for Catalyst.

ECMA TC33/TGRM Reference Model: In the "Document History" section of the ECMA Reference Model[6], the author, Anthony Earl states "When the ECMA Technical Committee for PCTE standardisation (sic) was formed, one of the decisions was to aim to create a CASE environment framework Reference Model to assist the standardisation (sic) process. A Task Group (TC33/TGRM) was formed in 1988 to develop a complete Reference Model. During 1989 the first full version of the Reference Model was created...".

Before the written ECMA Reference Model definition was distributed, the graphical illustration of the overall Reference Model structure[7] began appearing in environment builder presentations. In these presentations, the environment builders used this graphical illustration (now known as the "toaster model") to define those ISEE services which are supported by their specific environments. In these presentations, every toaster model component which was in some manner supported by the subject environment was highlighted. The degree of support provided by the environment, was indicated by the degree of highlight applied. In order to evaluate which environment best meets the requirements of a particular user, consumers were encouraged to simply compare the various toaster model representations. On the surface this seems like a great idea; something that all users can easily understand. However, the phrase *"buyer beware"* still applies when using these graphical environment service mappings. Because these environment builders either did not have the written definition of what ISEE services were represented in each of the toaster model's graphical components or they did not have a common understanding of how their specific environment's services related to the ECMA definition, what results in trying to compare the various toaster model mappings is often an "apples to oranges" situation. Now that the ECMA Reference Model document itself has been published, some of the earlier environment mapping inconsistencies may be overcome.

Overall, the ECMA TC33/TGRM Reference Model is a very valuable source of information for future ISEE standard efforts. As stated by Anthony Earl in the "Document History" section, of the ECMA Reference Model[6], "It should be understood that the reference model is totally independent of PCTE and is not intentionally biased towards PCTE. The ECMA group see (sic) it as an aid in identifying future standards that will be needed in addition to PCTE. It is also a valuable way of describing, comparing, and contrasting CASE environment frameworks."

NIST ISEE Reference Model : In 1989, NIST began holding a series of ISEE workshops. The objectives of these workshops, as defined by William Wong of the NIST / National Computer Systems Laboratory (NCSL)[5], are to :

- Identify and explore fundamental issues in ISEE areas;
- Identify the needed set of standards that define a comprehensive interface for integrating software tools;
- Develop guidelines on interface standards for ISEEs; and
- Provide guidance to Federal agencies to acquire ISEEs.

At the NIST 2nd Workshop on ISEEs, a graphical reference model (Figure 1) was developed which the majority of the participants deemed to be accurately representative of the major ISEE components. This model identifies and isolates those ISEE component interfaces for which no standard exists and for which no standardization effort is presently underway.

NIST is utilizing this model to map all the various ISEE standards/definitions into the specific ISEE component(s) that they address. The textual description of the services provided by each ISEE component is presently being defined in the NIST Integrated Software Engineering Environment Reference Model. The baseline for the ISEE service definition is the SEE Framework Reference Model developed by the ECMA Task Group TC33/TGRM[6]. In the initial analysis of this definition, it was determined that TC33/TGRM's reference model primarily addresses the component identified in Figure 1, as the "Framework." Other ISEE components identified in Figure 1, and some additional functionality that is defined in the NIST ISEE Framework component, are not defined in the ECMA reference model. NIST ISEE workshop participants are continuing to analyze the ECMA document to determine modifications/additions which must be made in order to define a complete reference model for the ISEE concept represented in Figure 1. The resulting definition will be documented in the NIST Integrated Software Engineering Environment Reference Model.

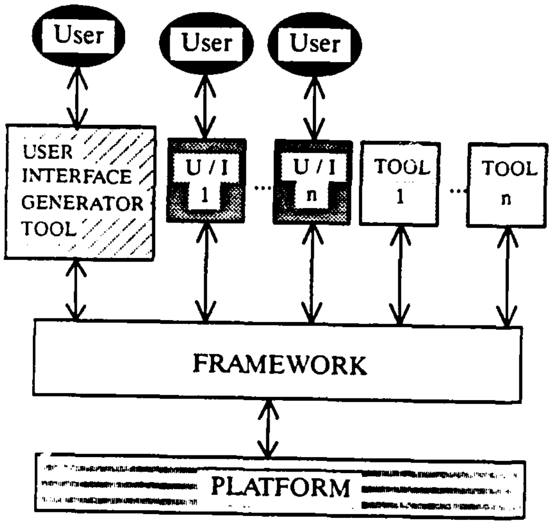


Figure 1 - Graphical ISEE Reference Model

STARS : The Software Technology for Adaptable Reliable Software (STARS) program is developing ISEE technology. This technology is deliverable to the government in 1994. The STARS prime contractors (IBM, Boeing and Unisys) are then free to continue development of the STARS ISEE technology and to market the resulting ISEE products to industry.

The question has been posed as to whether or not an individual corporation's efforts to develop an ISEE is a waste of time when compared with the Department of Defense (DOD) STARS effort, the amount of resources that are being expended in that effort, and the probability that DOD policy will mandate the use of STARS-developed ISEEs, once they are delivered.

The STARS charter, as documented by Dr. John F. Kramer in the STARS NEWSLETTER, Volume I, Number 1, dated May, 1990, does not mention the manner in which the STARS ISEE technology will be utilized once it has been delivered. However, looking at the DOD's historical approach for enforcing the utilization of government-developed technology, several possible scenarios may occur :

1. *Government contracts require contractors to use the environments developed under STARS for software development; thus, contractors have to acquire the environments from IBM, Boeing, Unisys or subsequent vendors.*

- **Disadvantage** - For most contracts, the cost to acquire the ISEE, to train project personnel into its use and to maintain the ISEE is tacked onto the cost of the contract, itself. Even with a significant price discount, this situation places the government in the position of paying at least twice for the ISEE; once for its development and then again for its acquisition by a specific project.
- **Disadvantage** - The usage of an ISEE is needed now. It is not economically nor competitively feasible for a contractor to wait until a STARS-developed ISEE is available to begin improving their software development practices. The Software Engineering Institute's Software Process Assessment Project has made contractors very aware of how their marketability will be affected by delaying the definition and implementation of corporate improvements supporting better software engineering practices.

2. *Government contracts require contractors to use a STARS-developed ISEE for software development; contractors receive the ISEE as government furnished equipment.*

- **Disadvantage** - For most contracts, the cost to install the ISEE, to train project personnel into its use and to maintain the ISEE is, again, tacked onto the cost of the contract itself. Again, the government is, in some manner, paying twice for the ISEE.
- **Disadvantage** - The previously described problems associated with corporations delaying the acquisition and usage of an ISEE, also apply to this situation.

3. *Government contracts require contractors to use an ISEE which is capable of achieving data interoperability with the specific STARS-developed ISEE utilized by the government agency sponsoring the project.*

- **Advantage** - In this scenario the government avoids any additional contractual ISEE costs, and the contractors can act now to implement their specific, corporate software engineering practice improvements.
- **Disadvantage** - This scenario could present a problem for the government in that a specific contractor's ISEE may not function as stated by the contractor, and may cause some difficulties in delivery or maintenance. However, this disadvantage has a positive solution. In order to avoid this problem, the government should apply (or have the contractor apply) assessment technology to the proposed ISEE to determine its functionality and adherence to standards. There are many documents/studies which may be utilized to evaluate an ISEE. Specifically, the Reference System, version 2.0, developed by the Ada Joint Program Office Evaluation and Validation Team; IEEE P1209 "The Recommended Guideline for the Evaluation and Selection of CASE Tools"; "A Reference Model for Computer Assisted Software Engineering Environment Frameworks" developed by the European Computer Manufacturing Association

(ECMA) Task Group TC33/TGRM; and the "NIST ISEE Reference Model" and the "NIST ISEE Reference Model Mapping Guidelines" documents developed by the NIST ISEE work shop participants.

Part of the STARS charter is to stimulate community development of ISEE technology. With this purpose in mind, in any of the above scenarios, an individual corporation's efforts to develop an ISEE benefits both the STARS program and the individual corporation. Within the STARS program, corporate ISEE development can be viewed as an indicator of successful ISEE technology stimulation. In-house ISEE development serves the specific needs of the corporation while enhancing the corporation's competitive stance. Thus, individual corporate ISEE development efforts that comply with ISEE community standards, should not be viewed as interim solutions that will be replaced by STARS-developed ISEEs. Because of this, the Catalyst and STARS ISEEs efforts should not be thought of as competitive, but should be viewed as complementary efforts.

If the DOD chooses to adopt a policy which implements STARS-related technology in the manner described in the third scenario, Catalyst's planned growth should ensure that it will be capable of communicating with a STARS-developed ISEE and that a good degree of data interoperability between the STARS-developed ISEE and Catalyst will be achieved.

ISEE Standard Selection : The ideal situation for environment builders is for all ISEE standards groups to coordinate their efforts. In the past, there have been some "territorial boundaries" which have worked to prevent any great degree of coordination from happening. However, today's ISEE community is paying heed to the "lessons learned" of the past and is working very hard to overcome any obstacles which prevent the coordination of good technology. Individuals of the ISEE community are also looking beyond their specific interests to ascertain that their technology is complementary to associated technologies and that they are not, in some manner, "re-inventing the wheel." Because of the enormous amount of resources required to perform all the work involved in the ISEE area, it is to the benefit of everyone to make this happen. There is evidence that this coordination of effort has begun.

Because they have common goals, the ECMA and NIST ISEE efforts have made a commitment to assist each other in the establishment of a standard ISEE reference model. Beginning with the 3rd NIST ISEE work shop, STARS prime contractors began, and have since continued, active participation in the NIST ISEE Reference Model effort. In late 1990, William Wong of NIST/NCSL made several presentations concerning the NIST ISEE work to groups involved in ISEE standard development (e.g., IWCASE, AIAA, etc.). The response he received from these presentations indicated a great community desire to coordinate ISEE standardization efforts. It is expected that these groups will either participate in or utilize the information that is created by the NIST ISEE effort.

After reviewing the NIST ISEE, ECMA TC33/TGRM, STARS and other related efforts, the emerging ISEE standard selected for use in Catalyst is the NIST ISEE Reference Model. In order to ensure Catalyst's compliance with the NIST ISEE Reference Model, Rockwell is taking an active role in the development of this standard; participating as a member of the NIST ISEE steering committee and chairing the Mapping Guidelines Working Group.

Evolutionary ISEE Development.

As previously seen in Figure 1, two of the more significant aspects of an ISEE are the separation of the user interface from the actual functionality of a tool and the establishment of a framework through which all tools and users (through their user interfaces) communicate.

Because of the internal requirement to immediately field a SEE which enhances software development productivity, increases product quality and encourages product artifact reuse, the first version of Catalyst (completed in December 1990) does not fully embody these two ISEE characteristics. The functionality of Catalyst, version 1, is more accurately represented by Figure 2, below.

As stated earlier, one of the requirements for Catalyst is to maximize the use of COTS products. This allows in–house environment development efforts to be focused on issues such as process, integration, automation, management, verification and availability.

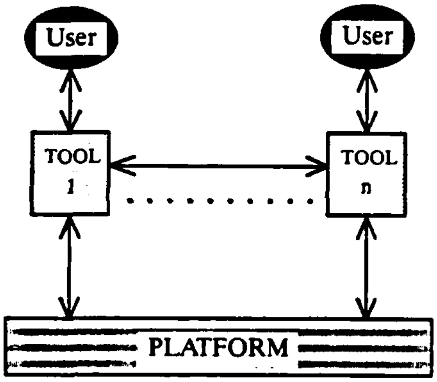


Figure 2 - Catalyst, version 1

When COTS products are used, there are three ways in which to get Vendor A's product to understand the data from Vendor B's product (i.e., data interoperability). The first method is the most desired, but at present is the option that is least likely to be achieved: *both Vendor A and Vendor B's products support the same data interchange standards.* The second method is not as desirable as the first, and in some cases is impossible to achieve, but has a history of working: *Vendors A and B cooperate with each other to develop a mechanism by which their products can communicate and achieve data interoperability with each other.* The third method is the least desired, but in many cases is the only option available: *the environment builder creates tools that act as interfaces, translators or filters between Vendor A and Vendor B products.* Rockwell has been very fortunate in that the vendors selected for Catalyst have, for the most part, supported the use of method 2, above, and have participated in the achievement of tool–to–tool communication, to varying degrees of success. Very limited use of method 3 has been applied in Catalyst, version 1. In the future, vendor product enhancements should allow environment builders to implement (the highly desired) first method, in their utilization of COTS products.

In method 2, because the vendors are supporting the communication of their respective products, that communication is guaranteed to be supported through all product enhancements; a direct benefit to the environment builders. This method also places the impetus on the vendors to provide support for data interchange standards within their products. The use of these standards is a very important key to the success of the next step in Catalyst's development: the introduction of the 'framework' ISEE component and the creation of an environment user interface.

Catalyst, version 2, will remove the tool–to–tool interface and replace it with the tool–to–framework interface. It will also remove the tool–to–platform interface and replace it with a framework–to–platform interface. The implementation of an environment user interface that separates the users from direct communication with the tools, will be the first step towards the full implementation of a user interface generator tool. These major development goals for

Catalyst, version 2, are depicted in Figure 3, below.

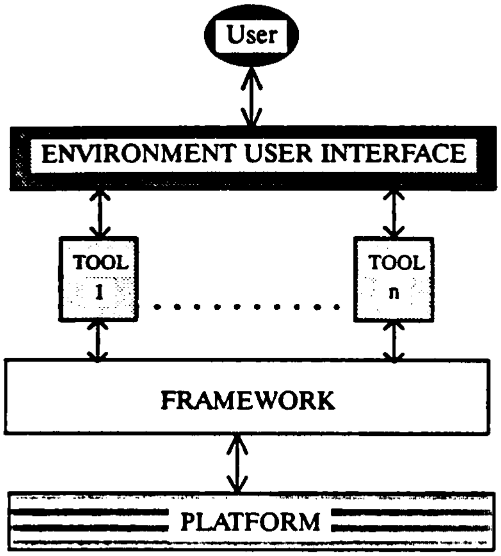


Figure 3 - Catalyst, version 2

In order to further facilitate the ease of transition from tool-to-tool to tool-to-framework communication, all tools which were created in-house for Catalyst, version 1, perform data identification, manipulation, verification and documentation based on the specific process definition and object classification that pertains to that data. Some of the more significant tools created for Catalyst, version 1, are the Requirements Management utility, the Project Unique Identifier utility and the Automatic Documentation Generation toolsets.

With Catalyst, version 2, in place, the next evolutionary step will be to implement the user interface generator tool and to transition the environment user interface to the control of that ISEE component (shown in Figure 1). This step also requires the transitioning of the COTS products' user interface control, to this ISEE component.

Because of the technical and societal complexities involved in achieving this ISEE characteristic it is unclear whether or not this will be directly achievable in a "near" timeframe. If not, Catalyst, version 3 will embody the maximum degree of support possible for this ISEE characteristic.

Catalyst's evolutionary environment development plan allows Rockwell's projects to immediately benefit from the use of a SEE and supports Catalyst's continued growth towards full implementation of a standard ISEE concept.

As Catalyst is growing towards a full ISEE concept, it will also be growing in terms of capability. The present functional status and availability of Catalyst, version 1, is discussed below.

Capabilities of Catalyst, version 1.

The implementation of characteristics that serve to identify a collection of tools and/or utilities as an ISEE are important to environment builders. However, for the most part, they don't impress environment users (unless those ISEE characteristics get in the user's way and then they are negatively impressed). For this reason, it is important to discuss the specific software development capabilities supported by Catalyst that are available to the user. The following identifies Catalyst's host platform availability and the specific software development life cycle activity capabilities supported in Catalyst, version 1.

Host Platform Availability : The pilot host platform for Catalyst, version 1, is the Apollo workstation utilizing $\text{UNIX}_{\text{bsd4.3.}}$ From the Apollo platform, Catalyst was ported to the SUN Sparcstation, again utilizing $\text{UNIX}_{\text{bsd4.3.}}$ Leaving the UNIX world, Catalyst was then ported to the DEC VAXStation 3100 utilizing VMS. The third environment port is planned for the DEC DECStation utilizing Ultrix.

Life Cycle Activity Support: The following identifies the major software development life cycle activity capabilities supported by Catalyst, version 1.

1. ***Full support for the software requirements analysis activity; specific capabilities are :***

- Requirements Engineering.
  - Verification
    (consistency, correctness, completeness and static performance)
  - Automatically documented in SRS and IRS
- Requirements Management.
  - Verification
    (consistency, correctness and completeness)
  - Tracking / Traceability
    (CSCI and System levels)
  - Automatic documentation in SRS
    (CSCI and System levels)
- Requirements Qualification.
  - Verification
    (consistency, correctness and completeness)
  - Automatically documented in SRS
    (CSCI level only; System level is documented in the Software Test Plan)
- Project Management.
  - Verification of process usage
  - Automatic, dynamic generation of SRS and IRS supports incremental, frequent generation of documents which is utilized for project progress tracking
  - Support for Change Impact Analysis :

    All source/destination occurrence pairs of data elements and interfaces are automatically and dynamically generated and documented in the SRS.

    All possible context views of requirements are automatically and dynamically generated and documented in the SRS on both a CSCI and System-level basis.

2. ***Full support for preliminary design activity; specific capabilities are :***

- Preliminary Design Engineering.
  - Implements process that supports projects of all sizes and any development language
    (doesn't fall apart when project grows or shrinks and isn't so specific in its support of certain language features that it is rendered useless or inefficient for use with languages that don't support those features)
  - Automatically documented in SDD, vn 1, and IDD
- Requirements Transition.
  - Controls requirements transition from software requirements analysis activity to preliminary design activity
- Requirements Management.
  - Verification
    (consistency, correctness and completeness)
  - Tracking / Traceability
    (development objects and document objects)
  - Automatic documentation in SDD, vn 1
- Project Management.
  - Verification of process usage
  - Automatic, dynamic generation of SDD, vn 1 and IDD supports incremental, frequent generation of documents is utilized for project progress tracking
  - Support for Change Impact Analysis :

    The controls placed on requirements transition facilitate the identification of the extent of impact, upon receipt of a change request.

3. ***Full support for detailed design activity; specific capabilities are :***

- Detailed Design Engineering.
  - Implements a process which supports projects of all sizes and which are developed in either Ada or C
    (the reasons for the present language limitation in the detailed design activity, are mostly conceptual)

- Automatically documented in SDD, vn 2 *(presently under development)*

- Requirements Management.
  - Verification *(consistency, correctness and completeness)*
  - Tracking / Traceability *(development objects and document objects)*
  - Automatic documentation in SDD, vn 2 *(presently under development)*

- Project Management.
  - Verification of process usage
  - Automatic, dynamic generation of SDD, vn 2, supporting incremental, frequent generation of documents which is utilized for project progress tracking
  - Support for Change Impact Analysis :

    The Requirements Management process facilitates the identification of all objects impacted by a specific change request.

4. ***Support for code/unit test activity; specific capabilities are :***

- Uniform code creation.
  - Provides common native code generation for Ada source
  - Allows each project to insert the specific target code generator required by their project

- Requirements Management.
  - Verification *(consistency, correctness and completeness)*
  - Tracking / Traceability *(development objects and document objects)*

- Automatic Generation of "White Box" test cases *(based on CSU and/or CSC definition).*
  - Facilitates "cleansing" and verification of code prior to moving to target test environment

- Project Management.
  - Verification of process usage
  - Automatic generation of Software Development Files (SDF) and SDF artifacts, facilitating configuration management
  - Support for Change Impact Analysis :

    The Requirements Management process facilitates the identification of all objects impacted by a specific change request.

Catalyst Development Process Definition :

In order to achieve the amount of automation, documentation and verification that has been accomplished in Catalyst, version 1, a development process had to be established which would result in the uniform, consistent development of high quality project artifacts. In developing this process, engineering, testing and documentation objectives were reviewed to assure that the defined process did not ignore requirements for the accomplishment of any particular objective. From this effort, a preliminary development process was defined. This process definition was then analyzed from the context view of both commercial and government projects. What was discovered is that the engineering objectives for commercial and government projects are essentially the same. Some differences exist in the testing objectives due to the application of more or less rigorous verification requirements. However, it was determined that most of the differences existing between commercial and government projects occur in the area of documentation. Upon closer review, it was found that the differences in this area centered more in the area of document format rather than content. Because the proposed development process identifies and manipulates data based on its object classification, the process was deemed to exhibit the qualities of flexibility and tailorability, and was, thus, appropriate for use in both commercial and government projects.

Next, the proposed process definition was analyzed to determine its applicability to multiple application domains, sizes and source language implementations. After ensuring that the process provided support for all these issues, the definition was finalized and adopted as the Catalyst software development process. This process is documented in the Catalyst Environment Reference Manual. Projects that utilize this development process gain the full benefits of Catalyst. Projects that partially utilize it, or are too far into development to begin use of Catalyst, still benefit from the many "stand-alone" capabilities supplied by Catalyst.

A significant benefit that has already been demonstrated by several Rockwell projects is that even partial usage of Catalyst by one project results in the opportunity for project artifact reuse by other projects. This opportunity for software development artifact reuse has been taken many times by new Rockwell projects; and has, in turn, encouraged those new projects to utilize Catalyst in their development.

It may seem strange that projects would have to be "encouraged" to use a technology that has been proven to save development time and increase product quality. However, the introduction of new technology, no matter how beneficial it is, has direct financial and schedule impact on each project trying to incorporate it.

Conclusion

Even though, Catalyst, version 1, was completed in December 1990, it has actually been utilized for production purposes since January 1989, by various Rockwell projects (multiple applications, sizes and languages). While the difficulty in building the environment was magnified due to the need to provide constant support to actual users during its development, the experience also provided an invaluable "test" of everything associated with the environment (i.e., process, capabilities, capacities, etc.). The immediate feedback received from the users (positive and negative) helped shape the environment into a technology that has been proven to support the real needs of projects.

The "willingness" that was exhibited by most project personnel and managers to make the effort to learn and implement this new technology (in any way possible and at whatever the cost), is further evidence that the technology meets some of their major development needs. The cooperation received from project personnel using Catalyst while the environment builders were trying to enhance it, was greatly appreciated by the environment builders.

After two years of experiencing the pitfalls associated with inserting ISEE technology into corporate production efforts, it is felt that the technology transfer procedures that have been established for Catalyst, while not perfect, do achieve some significant success in technology transition.

References

[1] IEEE Computer Society's Task Force on Professional Computing Tools, "(Draft) A Standard Reference Model for Computing System Tool Interconnections", October 5, 1990.

[2] C. Colket, PCIS Effort Gets Reviewers, Expert Team, *Ada Information Clearinghouse Newsletter*, December 1990.

[3] U.S. Department of Defense, *Defense System Software Development*, DOD-STD-2167A, 29 February 1988.

[4] H. Romanowsky, "Ada Programming Support Environment Development Plan", Version 1.0, August, 1985.

[5] W. Wong, "Summary of the 4th Workshop on Integrated Software Engineering Environments (ISEE)", October 30, 1990.

[6] A. Earl, ECMA Task Group TC33/TGRM's "A Reference Model for Computer Assisted Software Engineering Environment Frameworks", version 4.0, August 17, 1990.

[7] A. Earl, ECMA Task Group TC33/TGRM's "A Reference Model for Computer Assisted Software Engineering Environment Frameworks", version 4.0, page 11, Figure 1, August 17, 1990.

Biography

Sandra L Mulholland
Technical Staff Member
Rockwell International
Collins Commercial Avionics
MS 124-211
400 Collins Road NE
Cedar Rapids, IA 52498
(319)-395-4047

Ms. Mulholland is Project Engineer for the Catalyst Life Cycle Toolset. Ms. Mulholland has been very active in the Ada and environment technology communities. Ms. Mulholland is a Technical Advisor to the Software Productivity Consortium's Integrated Software Engineering Environment project (1990-present); an invited participant of the National Institute of Standards and Technology Integrated Software Engineering Environment Work Shop series (1989-present); a member of the NIST ISEE steering committee; chairperson of the NIST ISEE Reference Model Mapping Guidelines working group; a selected member of the Portable Common Interface Set (PCIS) Expert Review Team; a member of the IEEE Standards Working Group P1209; and a member of the ACM/SIGAda Software Development Standards and Ada working group. Ms. Mulholland has participated as an invited expert panelist on the "Joint Integrated Avionics Working Group Proposed Ada 9x Changes" panel sponsored by the AdaJUG Embedded Computing System Ada Issues Working Group (July 1989) and on the "Software First" panel sponsored by Tri-Ada (October 1989). From September 1985 until its disbandment in September 1990, Ms. Mulholland was a Distinguished Reviewer for the Ada Joint Program Office, Evaluation & Validation Team participating specifically in the Requirements Working Group, the Classification Working group, the Ada Compiler Evaluation Capability Working Group and the review of the AJPO Quality Testing Service plan. Ms. Mulholland received a Bachelor of Arts from the College of Natural Sciences at the University of Texas at Austin in May 1983.

TRANSFORMING THE 2167A REQUIREMENTS DEFINITION MODEL INTO AN ADA-OBJECT ORIENTED DESIGN

Joseph T. Lukman

Contel Corporation, Government Systems Group
31717 La Tienda Drive, Westlake Village, California, 91359

Abstract

This paper defines how to derive an Ada-object oriented design from DOD-STD-2167A. The requirements specification and preliminary design of DOD-STD-2167A use data flow diagrams and supporting data dictionaries. Guidelines are given to derive objects from data flow diagrams and data dictionaries, and to implement these objects in Ada. DOD-STD-2167A is tailorable to identify the system objects, while still maintaining functional decomposition. The software requirements specification can be expanded to define objects, and document the data and functional requirements distributed to them. An addition to the preliminary design section of the software design document can define the high-level design of the entities including data element identification and behavioral description.

Introduction

The cost and complexity of developing software systems has risen dramatically during the past fifteen years. To combat the rising financial burden and to increase the life of the system, the Department of Defense (DoD) has turned to open system architectures, software modularity and transportability. In this environment, two standards are of particular importance to software developers. Documentation standard DOD-STD-2167A defines a uniform system development representation. DoD programming standard ANSI/MIL-STD-1815A defines a software implementation tool. These standards combine to modularize the software development life-cycle.

The documentation standard DOD-STD-2167A is applicable throughout the system developmenl life-cycle. The Scftware Requirements Specification (SRS) document specifies the engineering and qualification requirements for a Computer Software Configuration Item (CSCI).[1] The preliminary design section of the Software Design Document (SDD) combines requirements definition with Computer Software Unit (CSU) level design.[2] Both documents can define the software system requirements as Data Flow Diagrams (DFDs) with supporting data dictionary.

ANSI/MIL-STD-1815A specifies the programming language Ada.[3] Ada has been designated to be the programming language for all DoD software. Its purpose is to provide a standard programming language for all embedded computer system software.[4] Ada uses many of the programming language constructs inherent to the object oriented paradigm. If an object oriented software design is derivable from the SRS and preliminary design section of the SDD, then Ada programming is simplified.

This paper defines techniques for transforming the requirements definition model derived for DOD-STD-2167A into an object oriented design. The programming language Ada implements the object oriented design. Included is a brief introduction to the object oriented paradigm and Ada, followed by techniques for transforming DFDs and data dictionaries into an object oriented design. Suggestions are made for tailoring DOD-SID-2167A to better represent object oriented software development.

The Object Oriented Paradigm and Ada

The object oriented paradigm reduces software cost and complexity by increasing software reusability. The heart of the object oriented paradigm is the modularization of the system's processes and data. Abstract data types are modular program units which increase software reusability by providing an implementation independent interface to implementation dependent code. An abstract data type is the encapsulation of an underlying data type with a set of operations that act upon the data type.[5]

Abstract data types are realized through class definitions. A class defines both an underlying data type and a set of operations known as methods. The methods dictate the behavior of the class. A method is invoked to manipulate the underlying data type. Sending a message to an object including an object name, method name and parameter list, invokes a method. An object is the instance of a class.

The programming language Ada contains constructs implementing much of the object oriented paradigm. This report focuses on packages and generics, the Ada constructs well suited for realizing abstract data types.

Package program units normally consist of a specification part and a body part.[6] The specification part defines elements of the package visible to the outside world. Such definitions might be subroutines (both functions and procedures), constants, data elements, etc. The body part defines how the package operates by implementing the subroutines defined in the specification part. Elements of the body part not defined in the specification part are hidden from the software which use package.[7]

Generic program units are subroutines and packages containing partially specified parameters. The parameters are identified and the generic is invoked during compilation.[8] Since the generic may be a package, it is also used to implement objects.

Package and generic program units can implement abstract data types by hiding data elements within the package body. A data element is encapsulated within subroutines by hiding the data element and making the subroutines acting upon it visible. The visible subroutines correspond to methods in the object oriented paradigm.

2167A Structured Analysis Model

DOD-STD-2167A requires documentation generation throughout the system development life-cycle. The SRS and the preliminary design section of the SDD are of particular interest to software developers. They define the requirements and how the software will satisfy them. The SRS specifies the required activities and data elements of the CSCI. The preliminary design section of the SDD specifies at the CSU-level the CSCI design. The SRS and SDD accomplish these tasks through a decompositional view of the system which may include DFDs with accompanying data dictionary.

Data Flow Diagrams

DFDs support the CSCI decompositional view of the system requirements. They decompose the requirements of a system into a network of processes interconnected by data flows. DFDs describe the CSCI's external interfaces, data at rest within the CSCI and the processes required by the CSCI.

Section 3.1 of the The SRS and SDD detail the external interfaces of the CSCI. A context diagram graphically depicts these interfaces as data flows between the system and external entities. The context diagram identifies external entities as terminators. Terminators transform into objects implementable as Ada packages. A separate object encapsulates each of the system's terminators.

Interfaces between the system and its terminators can be encapsulated within objects. This is accomplished by encapsulating all device dependent calls between the system and a specific terminator within an object. The methods of the object provide a system independent interface to the terminator. Sending a message to the object extracts information from or gives information to a terminator.

Ada packages and generics can implement objects encapsulating terminators. Visible subroutines and variables provide a device independent interface between the software and the terminator. Hidden subroutines and executable code send information to and read information from the terminator. If the device dependent hidden interface between the system and the terminator changes, the device independent visible package interface does not.

Section 3.3 of the SRS and section 3.1.2 of the SDD detail the internal interfaces of the CSCI. DFDs graphically depict the internal interfaces as data flows between computer software components. DFDs identify data flows at rest as data stores. Data stores transform into the hidden data structure of objects implementable as Ada packages.

The object oriented paradigm views all data as objects and all objects as data. The object oriented paradigm therefore considers DFD data stores as objects. In transforming a data store into an abstract data type, both the data store and the processes manipulating the data store must be considered. The internal data structure of the object implements the data store. Processes manipulating the data store become the methods of the objects.

Ada packages and generics encapsulate DFD data stores with the manipulating processes. The data store transforms into hidden data structure of an Ada package. Visible package subroutines implement the processes manipulating the data store. Being a functional description, the processes describe the functionality of the package. The correspondence between a process and subroutine is not necessarily 1:1. Visible and hidden subroutines combine to realize the process's functionality. Thus, a simple process updating a data store might transform into one or many subroutines manipulating hidden data elements. The visible subroutines provide access and allow manipulation of the data elements.

Section 3.2.X of the SRS and section and section 3.1.2 of the SDD define the capabilities of the system. Execution and control definitions throughout different states and modes describe the capabilities. DFDs graphically represent capabilities as processes. Processes transform into many aspects of an Ada design including subroutines and packages.

A DFD process describes a required capability of a system. In many instances, a process transforms into the method(s) of an object. The method(s) implements the functionality of the object. In other instances, a process transforms into an individual object. If an input is equivalent to an output of a process, the process is transformable into an individual object.[9] The equivalent data flows become the internal data structure of the object within the functionality of the process.

Ada subroutines within a package or generic implement the functionality of DFD processes. A visible subroutine, either a procedure or function, operates on the input data of the process. A return value from a function, or parameters of a subroutine specified as having return value, implement the output from the process.

Ada packages and generics implement DFD processes transforming into individual objects. The hidden data structure of an Ada package implements the equivalent input and output data flow of the process. Visible and hidden subroutines combine to implement the functionality of the process.

Data Dictionary

Section 3.4 of the SRS dictates a data dictionary. The data dictionary provides information including a brief description, units of measure, etc, for each data element internal to the CSCI. It specifies the source and destination processes of each DFD data flow. Similarly, the data dictionary details each data flow in the context diagram. Since the object oriented paradigm considers all data as objects and all objects data, the data dictionary identifies objects for the object oriented design. Ada packages realize these objects.

The object oriented paradigm considers each data element of the data dictionary an object. Yet, an optimal Ada design would not have a distinct package for each data element. A rule of thumb approach is needed to help identify which data elements transform into Ada packages.

A simple approach to identify data structures which transform into Ada packages is as follows: note the compound data elements of the data dictionary. The compound data elements are data decomposing into multiple data elements. Of these, list the compound data elements most critical to the system's success. The judgement of the system designer determines the "criticalness" of an abstract data element. The output from this step is a list of compound data elements deemed critical to the success of the system. Convert these elements to objects.

The abstract data elements critical to the system's success are transformed into objects in an object oriented design. Ada packages realize the objects. The hidden data structure of packages implement critical data dictionary elements. Visible Subroutines provide access to the hidden data structure. Visible subroutines in an Ada package can incorporate DFD processes manipulating the critical data element.

Object Oriented 2167A

DOD-STD-2167A enforces no specific developmental methodology. It permits software design to be documented as object oriented. But tracing the requirements from a function oriented SRS to an object oriented SDD is difficult.[10] The difficulty reduces when the SRS is object oriented.[11] This presents a problem to many software developers who find it difficult to interpret DOD-STD-2167A as object oriented. For these developers, supplemental paragraphs in the SRS and SDD can be added to better represent object oriented software development. The paragraphs supplement a function oriented SRS and SDD to identify allocated requirements and high level design of system entities.

Software Requirements Specification

Entity identification in the SRS can be accomplished through a redistribution of the requirements definition model. DOD-STD-2167A views the requirements as a decomposed network of activities interconnected by data flows. This is a top-down requirements definition. Entity identification depicts the system as a compilation of distinct abstract data elements, each data element having precisely defined capabilities. The entity relationship model is a middle-out requirements definition. Object oriented system design and Ada programming logically realize a middle-out analysis. The following paragraphs suggest additions to the SRS to identify the data elements and capabilities of the system entities.

CSCI Entities. This Paragraph shall be numbered 3.3 and shall be divided into the following subparagraphs to describe the capabilities of the CSCI.

(Entity name) data elements. This subparagraph shall be numbered 3.3.X.1 (beginning with 3.3.1.1), shall identify the CSCI entity by name and shall specify the required data elements of the entity. Some or all of this information may be referenced and provided by section 3.4, CSCI data element requirements.

(Entity name) capabilities. This subparagraph shall be numbered 3.3.X.2 (beginning with 3.3.1.2), shall identify the CSCI entity by name and shall state the capabilities of the entity. Some or all of this information may be referenced and provided by section 3.1, CSCI external interface requirements, 3.2, CSCI capability requirements, and 3.3, CSCI internal interfaces. Entity relationship diagrams may be used to illustrate the structure and capabilities of the entity.

Required data elements allocated to the object are described in the data element section. The capabilities section describes required manipulations of the data elements. Both sections reference previously stated requirements, and their purpose is to identify the allocation of requirements to objects.

Software Design Document

The preliminary design section of the SDD details the CSU level design of the CSCI. This section allows the Government to view the overall software design without design details. To represent the design of object oriented software, an additional preliminary design section can add the high level design of system entities. The following paragraphs suggest additions to the preliminary design section of the SDD to define the high level design of system entities.

CSCI Entities. This Paragraph shall be numbered 3.3 and shall be divided into the following subparagraphs to describe the software entities of the CSCI.

(Entity name) data elements. This subparagraph shall be numbered 3.3.X.1 (beginning with 3.3.1.1), shall identify the CSCI entity by name and shall state the purpose of data elements of the entity. Identify and state each data element of the entity. The design information for data elements shall be provided in section 5.

(Entity name) capabilities. This subparagraph shall be numbered 3.3.X.2 (beginning with 3.3.1.2), shall identify the CSCI entity by name and shall state the purpose of the entity. This subparagraph shall identify the preliminary design of the entity. Some or all of this information may be referenced and provided by section 3.2, CSCI design description. Entity relationship diagrams may be used to illustrate the structure and capabilities of the entity.

The data element section describes the high level design of the internal data structure of the object. The capabilities section details the high level design of the objects subroutines. Both sections reference design information, and their purpose is the allocation of system design to software objects.

Conclusion

An object oriented design can be derived from the SRS and SDD preliminary design section of the DOD-STD-2167A. Ada implements an object oriented design. Ada programming is simplified by using the transformation techniques defined in this paper to derive an object oriented system design from a functional requirements analysis.

Both the SRS and preliminary design section of the SDD can be supplemented to reflect object oriented system analysis and system design, while still maintaining a functional decomposition approach. The SRS can be expanded to identify system entities and dictate the functional and data requirements allocated to them. An addition to the preliminary design section of the SDD can define the high level design of system entities, including both data element definitions and behavioral description. The additions suggested in this paper identify the requirements and design of system entities from functional descriptions.

Work to be done on this topic includes research in object granularity. What is optimal, a few big objects or a lot of small objects? What are the factors influencing object granularity? Also, each element of the data dictionary is considered an object. Can automatic methods be generated to accurately derive useful and necessary objects from the data dictionary? The object oriented approach is quickly gaining notoriety as a powerful software development tool. Advances in this field shall combat the software crisis by promoting open system architectures, software modularity and transportability.

Notes

1. The following document provides further information on the 2167A SRS: United States Department of Defense, DOD-STD-2167A, Data Item Description for Software Requirements Specification. Document Number DI-MCCR-80025A.

2. The following document provides further information on the 2167A SDD: United States Department of Defense, DOD-STD-2167A: Data Item Description for Software Design Document, Document Number DI-MCCR-80012A.

3. The following document provides further information on the Ada programming language: United States Department of Defense, Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A (1983).

4. Norman H. Cohen, Ada as a Second Language, (New York: McGraw Hill, 1986). Pg. 1.

5. Richard S. Wiener and Lewis J. Pinson, An Introduction to Object-Oriented Programming and C++, (Reading, Mass: Addison-Wesley Publishing Company, 1988). Pg. 1.

6. Serafino Amoroso and Giorgio Ingargiola, Ada: An Introduction to Program Design and Coding, (Marshfield, Mass: Pitman, 1985). Pg. 27.

7. Serafino Amoroso and Giorgio Ingargiola, Ada: An Introduction to Program Design and Coding, (Marshfield, Mass: Pitman, 1985). Pg. 28.

8. Serafino Amoroso and Giorgio Ingargiola, Ada: An Introduction to Program Design and Coding, (Marshfield, Mass: Pitman, 1985). Pg. 129.

9. Bruno Alabiso, "Transformation of Data Flow Analysis Models to Object Oriented Design", in OOPSLA '88 proceedings. Pgs. 340 - 341.

10. JLC/CSM (Joint Logistics Commanders / Subgroup on Computer Software Management), "Software Development Under DOD-STD-2167A: An Examination of Ten Key Issues", (October 25, 1989). Pg. 8.

11. JLC/CSM (Joint Logistics Commanders / Subgroup on Computer Software Management), "Software Development Under DOD-STD-2167A: An Examination of Ten Key Issues", (October 25, 1989). Pg. 8.

Joseph T. Lukman, 1060 Hendrix Ave.
Thousand Oaks, California, 91360

Joseph T. Lukman graduated from California Lutheran University in Thousand Oaks, Ca., with a Bachelor of Science degree in computer science, and from California State University at Northridge with a Master of Science degree in computer science. Recent areas of research have included image processing architectures, rapid prototyping and object oriented system design.

AN OVERVIEW OF THE CLEAR LAKE LIFE CYCLE MODEL (CLLCM)

Kathy Rogers
The MITRE Corporation
1120 NASA Road 1
Houston, Texas 77058
(713) 335-8568

Michael Bishop
Unisys
600 Gemini Avenue
Houston, Texas 77058
(713) 483-1753

Dr. Charles McKay
Software Engineering Research Center
University of Houston-Clear Lake
2700 Bay Area Boulevard, Box 447
Houston, Texas 77058
(713) 283-3830

Abstract

A system engineering life cycle model should be predicated on precise models which, in turn, are based on sound concepts and principles. This paper reviews the concepts and principles upon which the Clear Lake Life Cycle Model (CLLCM) is based and presents the architecture, standards, and disciplines that define the model. The paper also discusses the risks and benefits of the model. Emphasis is placed upon the model's unique strengths in support of managing change control[1] and integration across the life cycle. The model has been referenced by several programs and agencies including the Software Technology for Adaptable, Reliable Systems (STARS) Program, the Space Station Freedom Program (SSFP), and the National Institute of Standards and Technology (NIST).

Keywords: Life cycle model, change control, framework, integration, system engineering, software engineering environments, risk management, stable interface set.

1.0 Introduction

The Clear Lake Life Cycle Model (CLLCM) consists of a conceptual model of life cycle support environments and a proposed mapping to implementation models. This paper focuses on the logical properties of the conceptual model. A separate paper, [McKay, 1988], describes the issues involved in mapping physical properties of implementation models to logical properties of the conceptual model in a consistent manner.

The CLLCM conceptual model decomposes life cycle support environments into four major segments: platform, framework, tools and manual procedures, and stable interface sets[2]. The platform, shown in Figure 1-1, consists of a collection of hardware, operating systems, data base and file management systems, and data communications systems. The platform is encapsulated by a stable interface set which provides an integrated view of the underlying services and resources. The set of stable platform interfaces a.:o promotes the transportability and interoperability of the framework implementation. The framework is the key to the life cycle management of integration, change control, persistence, and traceability. The framework and its support for four levels of abstraction are believed necessary for future life cycle support environments, as will be discussed in the remainder of this paper. The stable interface set that separates the framework from both the tools and the sets of user interfaces is intended to be consistent with CAIS-A [CAIS-A, 198ᶜ] but is extended to provide a much finer granularity of semantic representation wherever needed. The heart of the framework is its ability to manage precise semantic models involving traceable sets of products, processes, and interfaces.

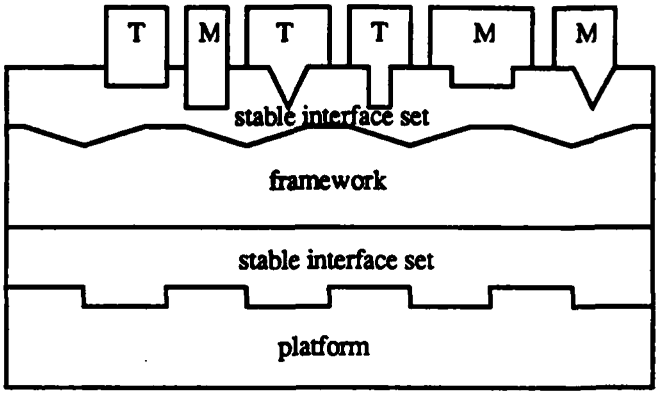
[1] Change control is used to describe a superset of what is usually considered configuration control [Muncaster-Jewell, 1988].

[2] The concept of a stable interface set must embody three characteristics:

1) an integrated view of the interface and its semantics,

2) information regarding the relationships between abstractions and their users (from these relationships come the extensibility rules), and

3) information regarding the provision of the classes of services and resources available as well as the exact steps necessary to extend those services and resources. It is the concept of providing precise rules and rigorous controls for extending an interface set which makes it stable.



T => Automated Tool
M => Manual Process

Figure 1-1. Major Segments of the Conceptual Model

2.0 Overview of Other Life Cycle Models

Existing life cycle models attempt to solve several life cycle problems, with varying success. The issues of maintaining and operating systems after acceptance, which are largely responsible for the term "software crisis", are well-known and will not be repeated here. An additional problem -- one which is amplified by existing life cycle models -- is the arbitrary distinction among the life cycle phases and the inherent ambiguity of the phases' end points. This section briefly describes the primary shortcomings of two popular life cycle paradigms, the waterfall life cycle model and the spiral life cycle model.

The waterfall life cycle model [Royce, 1970] is deficient in its ability to support the generation of large, complex software systems. Its policy of freezing life cycle artifacts, such as requirements and design documentation, far in advance of the system delivery date often prevents the creation and timely delivery of quality products. Figure 2-1 shows one rendition of a waterfall life cycle model. Notice that integration concerns are deferred until after coding. The waterfall life cycle, with its arbitrary life cycle phases, exhibits the "fuzzy end point" problem mentioned above. (In contrast, the Clear Lake Life Cycle Model emphasizes the processes and products of the life cycle rather than focussing on life cycle phases per se.)

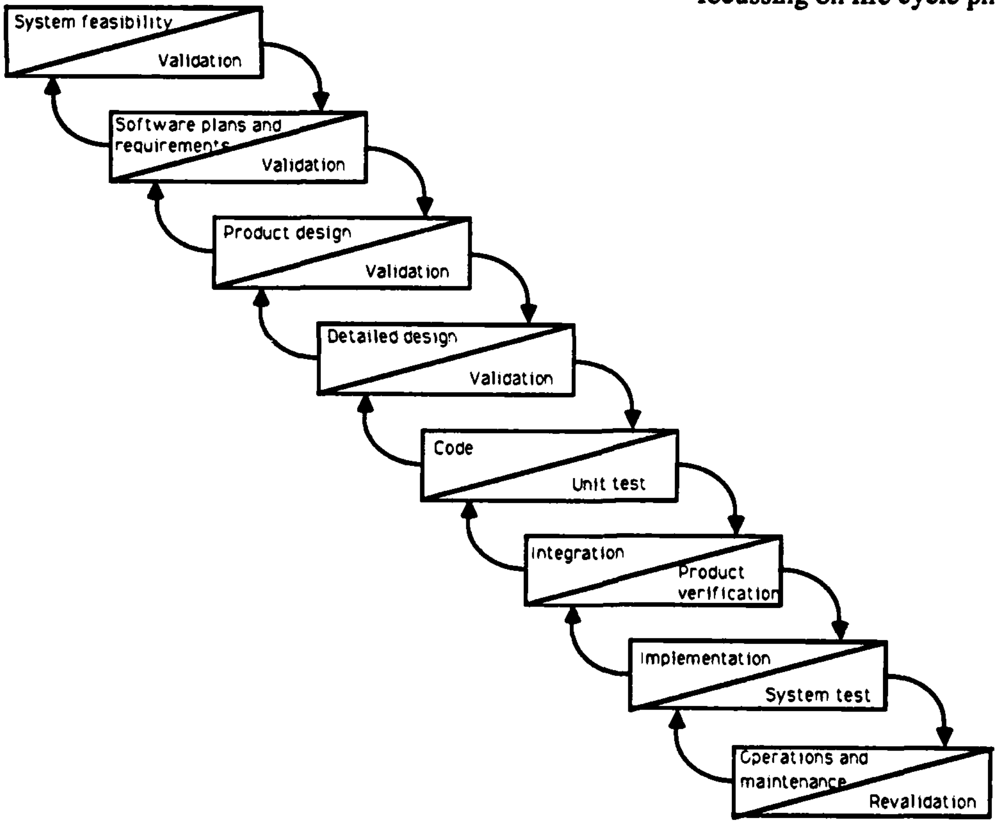


Figure 2-1 Waterfall Life Cycle Model

The spiral life cycle model [Boehm, 1988], on the other hand, is an approach in which artifacts are created and iteratively refined (with the involvement of contractor and client personnel) until they have evolved to a point at which they can be baselined. The spiral model is a much more realistic approach to software engineering than the waterfall model and is being adopted by some government organizations. The spiral approach facilitates the identification of problems and discrepancies early in the life cycle, when the cost of repair is far less expensive than it would be after deployment. Although it may be difficult for clients to accept an approach in which the majority of funds are allocated early in the life cycle, it would be preferable to the current state of software engineering, in which the emphasis shifts from the delivery of a quality system to the delivery of any system. Figure 2-2 illustrates Boehm's spiral life cycle model.

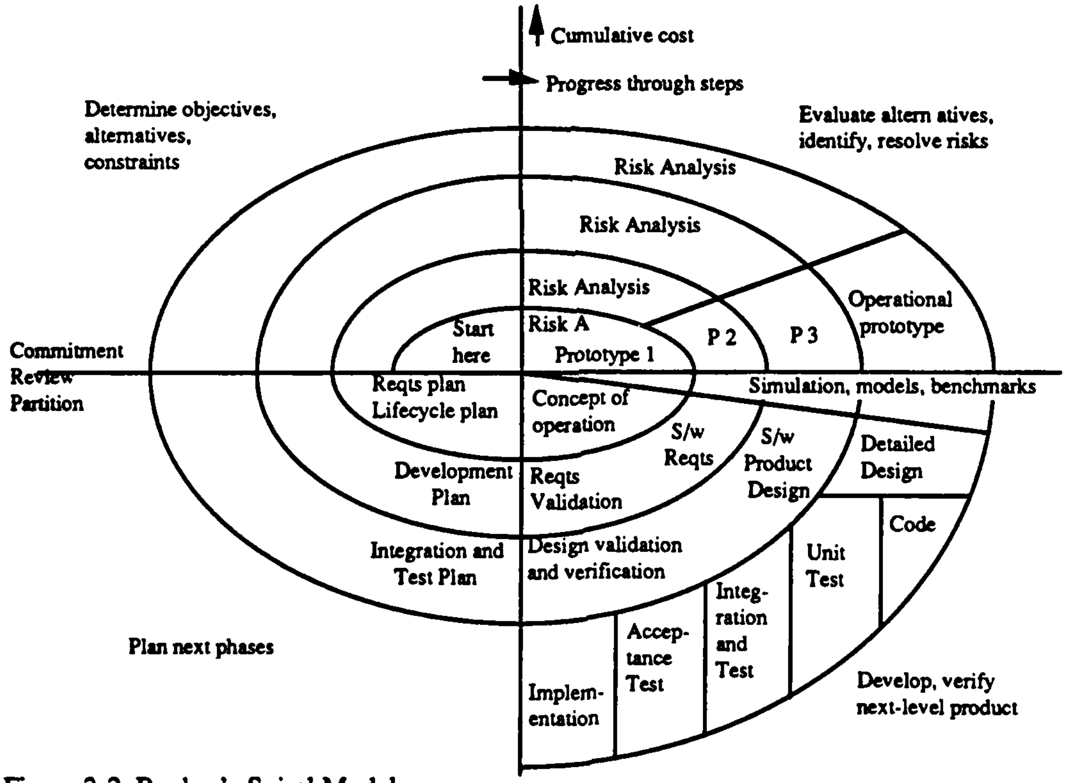


Figure 2-2 Boehm's Spiral Model

Although the spiral model is more appropriate (than the waterfall) for the large, complex, incrementally evolving projects addressed by CLLCM, the pervasive life cycle issues of integration and change control are not directly addressed by the spiral model. The CLLCM provides levels of abstraction, with precise models at and among the levels, throughout all phases and activities of the life cycle. These levels of abstraction are important for controlling complexity, managing risk, and facilitating integration in a manner appropriate to each level.

3.0 Perspectives Within the Clear Lake Life Cycle Model

There are four top-level agents involved in the system development process: the transformation team; the project and configuration management (PCM) team; the safety, reliability, and quality assurance (SR&QA) management team; and the automated support environment. Each has a somewhat different perspective when viewing the system development process, and each perspective must be supported by the appropriate information across the successive life cycle phases.

The transformation team is concerned with transforming information across the life cycle phases, in order to solve the problem being addressed by the system. The PCM team is charged with maintaining the schedules, budgets, and staffing plans, and the configurations of system components (hardware, software, and operator interfaces). This team must anticipate and deal with the changes that are inevitable in the development of a system. The SR&QA management team is concerned with verifying adherence to the process and validating the artifacts produced by the process. Tests and other assurance criteria must be specified and applied, and the results analyzed by the SR&QA management team, before an artifact is moved into the next life cycle phase. Automated support eases the manual tasks associated with the other three teams. Each of these four agents acts quasi-independently but with knowledge of, and input from, the others.

4.0 Conceptual Model of the Clear Lake Life Cycle

The Clear Lake Life Cycle Model is described by a conceptual model addressing the activities of each life cycle phase from the four perspectives mentioned in Section 3.0. The conceptual model describes the complex interrelationships among life cycle processes and products -- including the relationships among processes and the products they share -- and defines the key properties of these products, processes and relationships. The model provides a sound basis for leveraging reusability.

The CLLCM conceptual model emphasizes well-defined interfaces among and within the semantic models of the processes and products across the life cycle. This emphasis on "interface engineering" facilitates measurement and refinement of processes and products at varying levels of granularity. For example, refinement is possible within steps of a methodology (most appropriate for small changes between small interfaces) or alternate methodologies may be modeled and executed in parallel for comparison (recommended for large changes between large interfaces separated by many smaller interfaces).

The conceptual model of the CLLCM is characterized by four levels of abstraction that reflect the views of the client, technical and management personnel, library management, and object management. It is coincidental that there are four levels of abstraction -- they do not map directly to the four perspectives described in Section 3.0 (although they could be seen as representing the automated support environment's perspective in some sense). The following subsections describe the conceptual model's four levels of abstraction and the way in which each supports the three teams and the automated support environment.

4.1 First-Level Abstractions: The Client's View

Figure 4-1 shows the top level of the CLLCM conceptual model. This level represents the view of the client desiring that an automated system be produced. This view is depicted in terms of the system life cycle phases, the contractually required products (represented in the figure by the closed pairs of double parallel arcs) generated by the activities within each phase, and the process by which these products are verified, validated, and approved for inclusion in an evolving system baseline. The ellipse between phases P6, Testing, and P7, Deployment and Operation, is the acceptance test milestone. This milestone serves as the transition from system development to the maintenance and operation activity, which consists of iterations of the development phases of the life cycle. The project object base shown on the right-hand side of the figure supports the development and maintenance of the objects that serve as the basis for the products, artifacts (non-deliverable products created during the system engineering process), and processes involved in software engineering, hardware engineering, human factors engineering, operations, and logistics.

The first level of abstraction has the flexibility to support a waterfall life cycle model for small, simple, well understood systems, or a spiral model for larger, more complex systems. The mapping of this abstraction level onto a waterfall life cycle model is rather simple. The transformation activity (represented by the rectangles) is concerned with the generation of a product, the quality and safety management activity (represented by the circles) deals with the verification of the process and validation of the product, and the project and configuration management activity (represented by the pentagons) determines, upon receipt of a recommendation from the quality and safety management team, whether a product is to be accepted as a new member of the system baseline.

FIRST LEVEL ABSTRACTIONS

Conceptual Model of a Life Cycle Support Environment

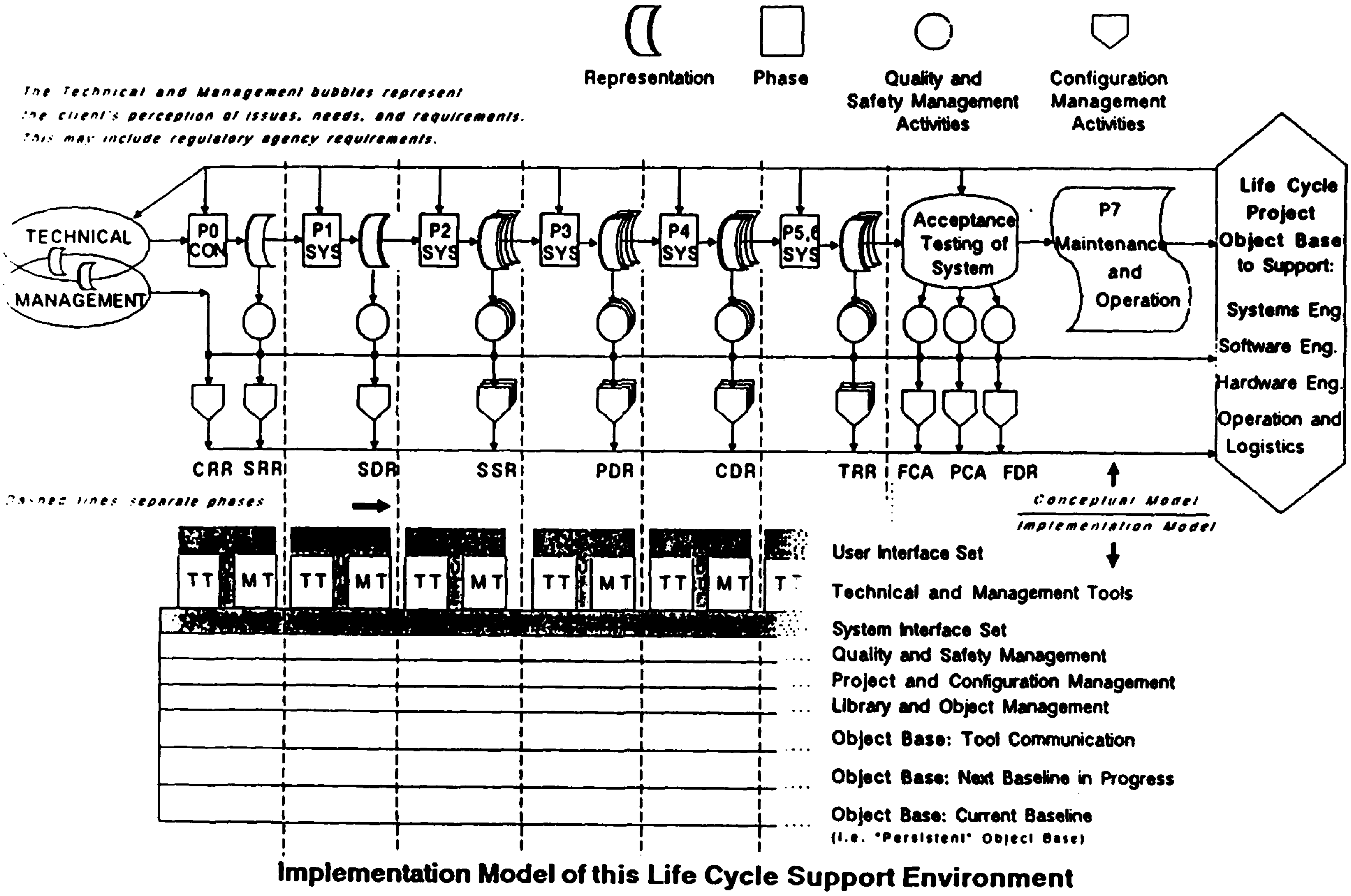

Representation
Phase
Quality and Safety Management Activities
Configuration Management Activities
The Technical and Management bubbles represent
the client's perception of issues, needs, and requirements.
This may include regulatory agency requirements.
TECHNICAL
MANAGEMENT
P0 CON
P1 SYS
P2 SYS
P3 SYS
P4 SYS
P5,6 SYS
Acceptance Testing of System
P7 Maintenance and Operation
Life Cycle Project Object Base to Support:
Systems Eng.
Software Eng.
Hardware Eng.
Operation and Logistics
CRR
SRR
SDR
SSR
PDR
CDR
TRR
FCA
PCA
FDR
Dashed lines separate phases
Conceptual Model
Implementation Model
TT
MT
User Interface Set
Technical and Management Tools
System Interface Set
Quality and Safety Management
Project and Configuration Management
Library and Object Management
Object Base: Tool Communication
Object Base: Next Baseline in Progress
Object Base: Current Baseline
(i.e. "Persistent" Object Base)
Implementation Model of this Life Cycle Support Environment

Figure 4-1 CLLCM First Level of Abstraction

The mapping of the top abstraction level onto a spiral life cycle model must address the additional activities included within the phases of the spiral life cycle. The transformation activity includes not only the generation of a product but the prototyping, simulation and modeling required to refine the product. The project and configuration management activity includes, in addition to baseline decisions, planning for subsequent phases and analysis of the risk involved with baselining or not baselining a life cycle product.

The client's view is only a portion of the total view that must be held by key members of the three teams and the automated environment that supports them. (This point will be amplified in Section 4.2, which describes the second level of abstraction.) The client's view is concerned with preserving a sense of the work flow as viewed by the client (as opposed to, say, the transformation team's sense of the work flow, which may be quite different). For any subcomponent of the work to be done, in any phase, the model assumes that the preceding phase has produced a baselined product to serve as an input stimulus for the transformation team. The response of this team is to produce an output artifact that is closer to the eventual form of the subcomponent. The subcomponent is intended to be integrated with other components and subcomponents, in order to pass acceptance testing and be deployed as part of the operational system. The SR&QA management team ensures that the output product and the process that produced it comply with project policies and standards. Members of the PCM team consider issues of budget, schedule, risk, and other implications for planning adjustments, before making decisions to advance the baseline for the subcomponent and other subcomponents and components in that phase.

4.2 Second-Level Abstractions: Management and Technical Views

Whereas the CLLCM conceptual model's first level of abstraction describes what is taking place during the system life cycle from the viewpoint of the client, the second level of abstraction describes what happens "behind the scenes" in order to support the client's view. Figure 4- 2 depicts the second level of abstraction. This level expands the information present in the first level in order to reflect the needs of the management and technical personnel who work with the life cycle support environment.

Two additional icons have been added at the second level of abstraction. The diamond represents relationships between entities on either side, where the entities represent processes, products, and artifacts. The closed pair of single parallel arcs represents artifacts which are needed within the environment but which are not contractually required (e.g., informal documents, checklists, questionnaires, interview transcripts, etc.). Each of the entities and relationships has attributes (properties) which provide information that must be preserved by the environment and its users. For example, an entity attribute may indicate that a product contains classified information. An attribute on the relationship between the classified product and a client representative may restrict the representative from providing information regarding the product from any place other than a secure processing site.

The sets of entities and relationships and their attributes comprise a semantic model in entity-attribute/relationship-attribute (EA/RA) form. This semantic model shows the organization of the major processes and products of the system life cycle and the relationships between them as well as the important properties of the processes, products, and relationships.

The reader should note that Figure 4-2 is far from complete. For example, as illustrated in the lower left corner of the figure, each process icon can be decomposed into a collection of icons. These icons represent approved methods, standards, tools, roles, etc. The methods may be further decomposed into steps and, in turn, into templates for achieving the desired results of the steps. In keeping with the recursive nature shown in the figure, each step or activity is stimulated by a product that represents the interface to the previous step (or activity). The recursive nature of the process permits the measurement and refinement discussed earlier in this section.

Another item of interest is that, within the second level of abstraction, each phase of the system life cycle from Requirements, P2, through Testing, P6, consists of a software activity, a hardware activity, and an operational interface activity. Each of these activities is staggered in time: in a given life cycle phase, the software transformation team

SECOND LEVEL ABSTRACTIONS

A Taxonomy of Taxonomies

The Life Cycle Support Environment has attributed relationships to all interfaces, phases, quality and safety activities, and project configuration management activities.

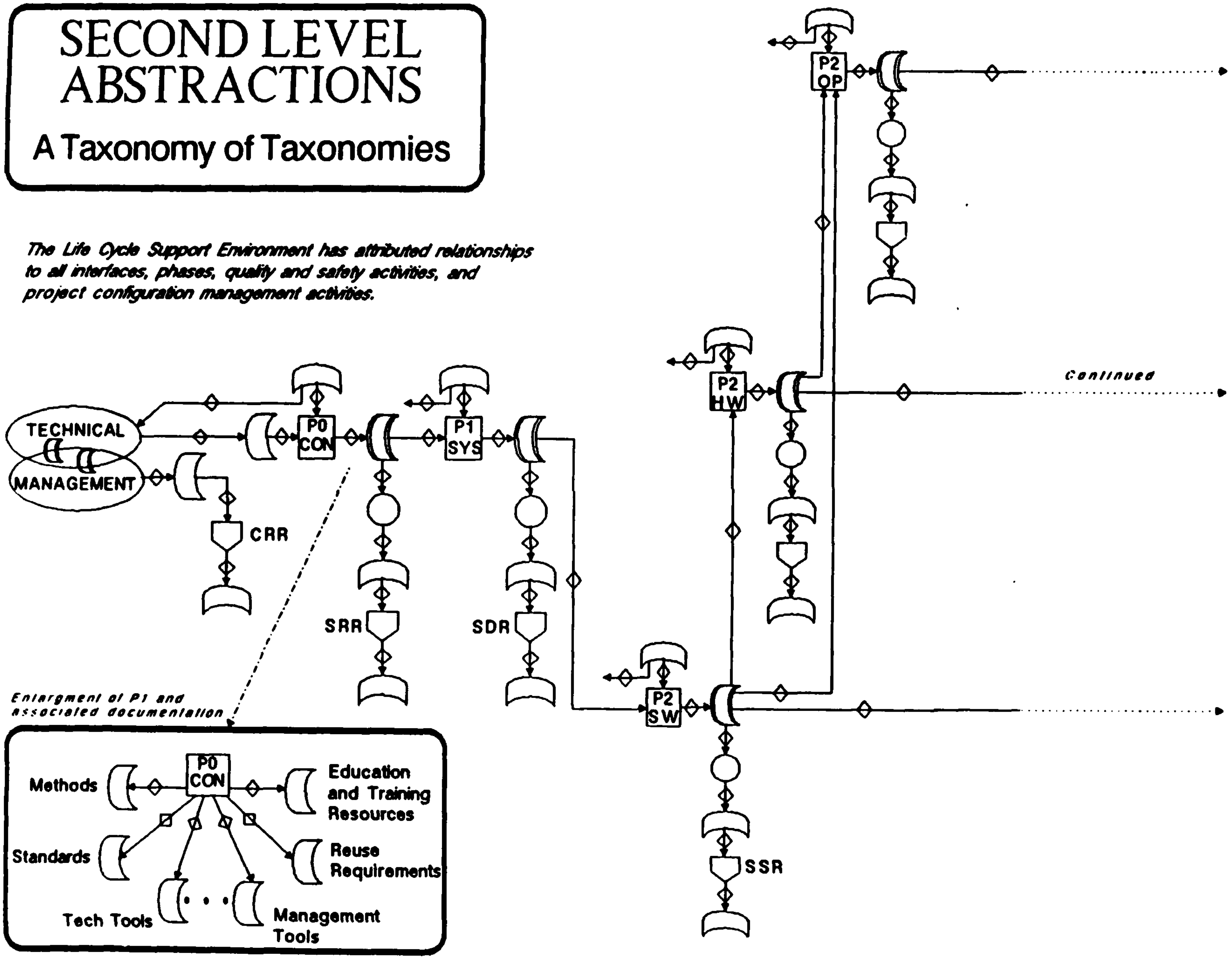


Figure 4-2 CLLCM Second Level of Abstraction

generates a product which is reviewed by the hardware transformation team, who then generate a hardware product supporting the software product. The operational interface transformation team acquires access to both the software and hardware products and generates an interface product, which then serves as the user interface to the software and hardware services. This multiple activity model provides a separation of concerns among the software, hardware and interface aspects of a system, facilitating change control by minimizing the effect that a change to one aspect of the system can have on the other two aspects. The model also emphasizes the required interactions and agreements that must be negotiated and enforced among the teams responsible for these three interdependent activities.

4.3 Third-Level Abstractions: Library and Component Management

The CLLCM conceptual model's third level of abstraction describes how the life cycle products shown in the first level of abstraction are organized for library management of components, views, and configurations. This level is depicted in Figure 4-3. The upper right corner of the figure shows a collection of actual objects. An actual object, such as A111 in the figure, could be any persistent object such as a component of a requirements document, a component of a design document, a schedule, a memo or an Ada generic package. Each actual object is associated with a unique primary entity upon creation; this primary entity contains name and type information concerning the actual object, and it is replicated in each library in which its actual object is needed. The attributes of the relationship between a primary entity and a copy of the primary entity describe the use of the corresponding actual object in the library in which the primary entity copy exists.

The upper left corner of Figure 4-3 shows how the third level of abstraction facilitates component reuse. The library of reusable components is described by a semantic model in which the entities are primary entities referencing actual objects and the relationships establish a classification scheme for objects in the reuse taxonomy.

The bottom half of Figure 4-3 shows how the library and component management model supports the generation of subsystems. Each component of the Network Communications Services (NCS) object (Virtual File Store (VFS), Virtual Terminal (VT), and Manufacturer's Automated Protocol (MAP)) is described by a semantic model showing the objects within the component and how those objects are integrated to form the component. The entities in the semantic models are copies of primary entities. The relationships between the primary entity copies and the originals indicate the use of the actual objects within the components in which they are referenced (e.g., instantiation parameters for a generic Ada package object). The primary entity copies are used to select and instantiate the objects needed to build the components within the Network Communications Services and, subsequently, the NCS itself.

The library and component management model also facilitates the configuration management of objects and object configurations. Each of the library configurations in Figure 4-3 (the reusable component library, the NCS and its components) is referred to as a stable framework (abbreviated as SF in the figure). A stable framework serves as a configuration management tool which shows exactly which objects are under baseline control within a component at a given point in time. Thus, stable frameworks and their associated semantic models provide the capability to assess the impact of adding, modifying or removing objects from stable components.

4.4 Fourth-Level Abstractions: Object Management

The fourth level of abstraction of the Clear Lake Life Cycle Model, depicted by Figure 4-4, describes an underlying representation of an object. This representation supports the management of issues such as component variations, revisions, releases, associated test sets and design rationale. The semantics of the representation are capable of capturing models of sets of information, associated behaviors, and their interfaces across all activities and phases of the life cycle.

At this fourth level of abstraction, an object is defined as a collection of entities connected by relationships. The entities represent object constituents such as public interfaces, bindings to runtime environments (also known as private interfaces), implementations, interface and implementation test sets, and rationale information.

THIRD LEVEL ABSTRACTIONS

Library and Component Management

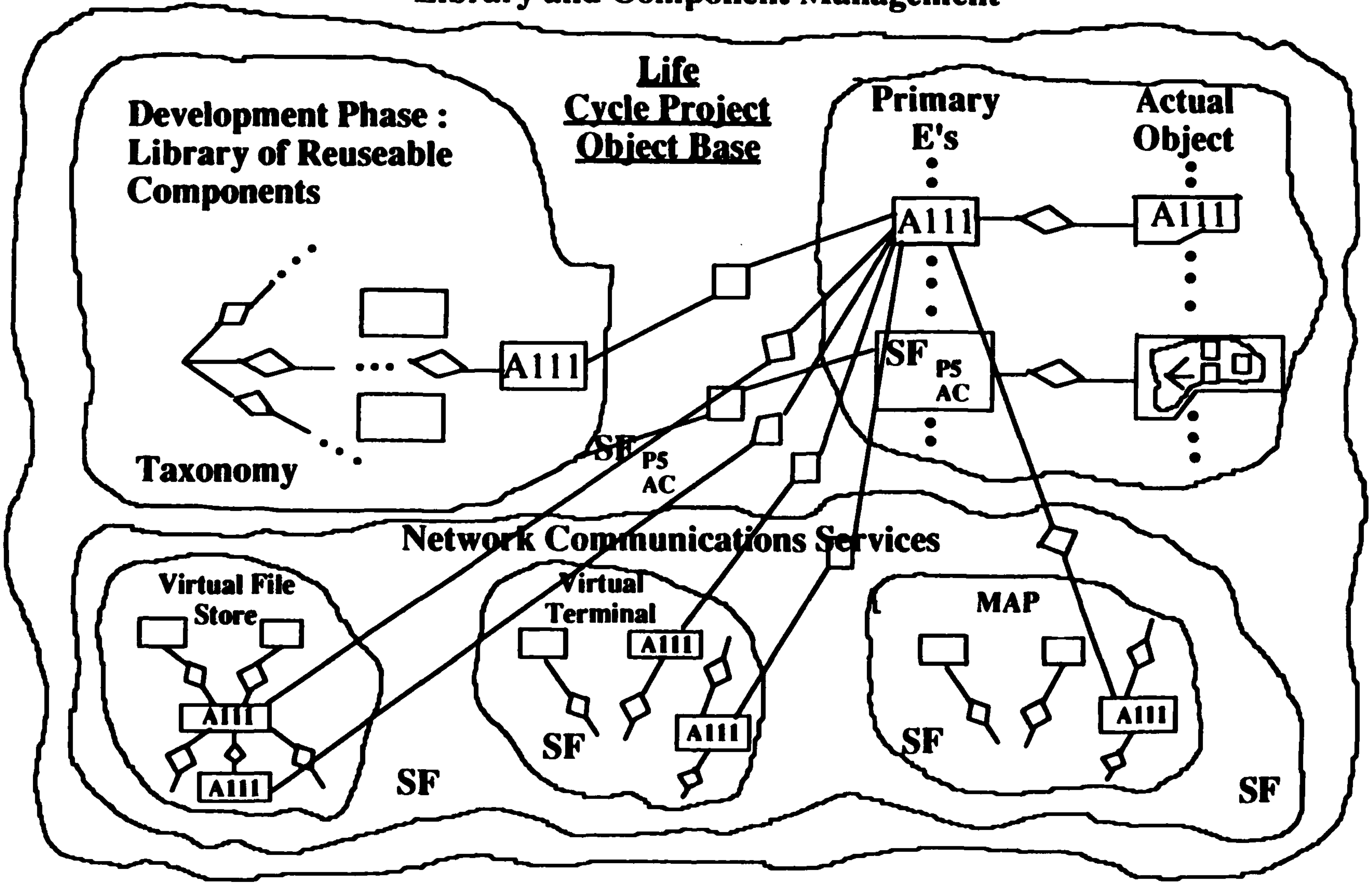


Figure 4-3 CLLCM Third Level of Abstraction

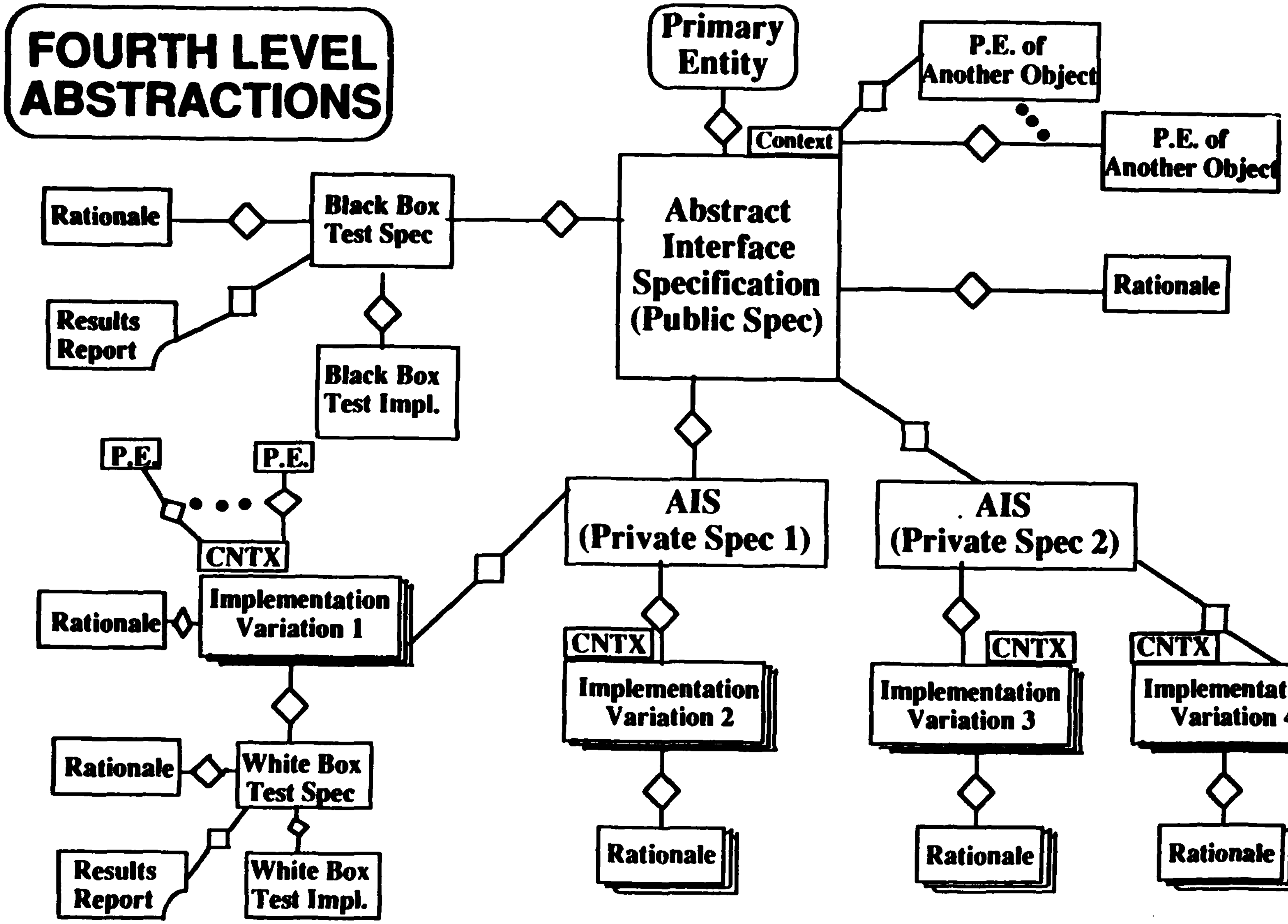


Figure 4-4 CLLCM Fourth Level of Abstraction

An object is referenced by a primary entity, as discussed in Section 4.3. The primary entity serves as the root of a path name which uniquely identifies the individual parts of an object. For example, a path name such as:

A111.Public_AIS.Private_AIS(Variation_1).
Implementation(Variation_2, Revision_2,
Release_1)

might represent the second revision of an implementation variation of object A111 which supplies software to emulate floating point hardware, bound by a private interface specification referencing the runtime environment of the host system (Variation_1). Release_1 signifies that this revision (2) was the first one released for use in configuration with other objects. The fourth level of abstraction supports multiple variations and revisions by allowing more than one private interface specification to be related to a public interface specification and more than one implementation to be related to a private interface specification.

These schema-level abstractions of the structure of objects support variable levels of granularity for the different views of life cycle activities and phases. Tools that require only a coarse granularity of representation (such as those conforming to the CAIS-A model) need not be aware of finer granularity representations, yet tools designed to exploit finer granularity semantics may do so. This support for variable granularity views of commonly structured objects (e.g., tools, artifacts, teams, schedules, processes, etc.) provides a unifying paradigm with many accruing benefits throughout the life cycle.

All objects, views, configurations, and libraries are actually stored as persistent objects in the form of the fourth level of abstraction. Some objects are considered to be leaf-level objects; examples include an Ada generic package, a role within a team organization, or any other object that is to be treated as a single building block with no need for subsequent decomposition by the user. Another type of object to be stored at the fourth level may be one that contains a directed graph depicting a collection of building block level objects and their relationships to one another in a given configuration. Such directed graph objects represent implementations at the third level of abstraction. The attributes of the objects and their relationships within a given configuration can be appropriately constrained by configuration objects, allowing them to be associated with predicate tests that activate triggers or daemons to enforce policies and help control processes within the project development environment. Such configuration objects represent an implementation of the second level of abstraction. An additional object that restricts visibility and determines access rights is stored as a view object. For example, the implementation of the first level of abstraction is represented as a view object that enforces the client's perspective of the realities stored at the second level of abstraction.

5.0 Risk Assessment and Management

Risk is associated with decisions and the interrelationships among those decisions. The activities of assessing, planning and managing risk are intrinsic to system development. Not all risks can be avoided, but they all should be assessed and explicit controls should be applied in proportion to each risk. The assessment of risk should lead to confinement of any identified risk so that it is isolated from the other phases of development, to the maximum extent possible. For each risk that has been identified and assessed, a management plan that reflects the explicit controls assigned to that risk should be developed. The assessment and control of risk should be iterated for each risk until the risk level has been reduced to a point at which it is within an acceptable threshold.

A system life cycle model should incorporate and support the assessment and management of risk throughout the life cycle. An approach based on Parnas' scheme of decomposing a problem space into non-overlapping segments is ideally suited for handling risk. With such an approach, a known risk can be confined to a single segment that is made as small and self-contained as possible -- that is, it is encapsulated by well-defined interfaces and reduced to its essence. Limiting an area of risk facilitates the isolation of the risk from the rest of the problem space, the identification of the risk to program management, and the assignment of a suitable team to address the risk and to repeat the steps of the assessment and control process to reduce the level of risk. Figure 5-1 depicts the problem space within the environment and the segmentation of that problem in order to address risk. To further facilitate risk

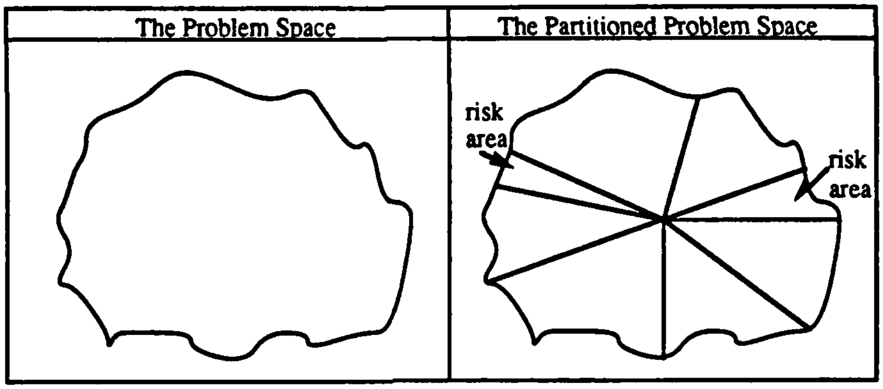


Figure 5-1 Partitioning the Problem Space

management, careful attention should be given to the semantic support needed by the four major perspectives of the life cycle (as enumerated in Section 3.0).

5.1 Potential Risks Associated With Use of the CLLCM

The risks involved with introducing new technology into large projects are numerous and critical. It is, therefore, necessary to evaluate the ways in which the model addresses risk as well as the specific risks introduced by the model. Where possible, the means for mitigating specific risks should be determined as well.

The major risks posed by the CLLCM model include the possibility that it could be assumed to implement only a "waterfall" life cycle process, the lack of extensive project experience in using the model, the need for automation of the processes within the model, and the need for appropriate library and object management technology to support the model. This section addresses the specific risks of using the model and; where known, suggested mitigating measures.

"Waterfall" Appearance: One risk of the Clear Lake Life Cycle Model arises from its resemblance to a traditional waterfall model. Even though the CLLCM diagram appears to depict a waterfall life cycle and can be used in that manner for simple projects, it is not limited to that paradigm. The activities of the waterfall life cycle model are used as the phases of the model because they characterize the basic activities involved in the development of any system. Although the phases are delineated by deliverables, that does not signify that a given phase will not be revisited. Also, different segments (and their increments of evolution) may proceed through the phases at different times.

The flexibility of the CLLCM allows it to be used in ways that are more risk-directed than the waterfall. The CLLCM is intended to address risk by identifying, evaluating, and mitigating risk at each step. In order to mitigate risk in all phases, the model is intended to provide feedback. Feedback can come from any phase into any other phase and is moderated through the object base, as depicted in Figure 4-1. The return arrow, near the top of Figure 4-1, indicates that as results of further phases are entered into the object base, feedback may indicate the need to revisit previous phases. As an example, information gained through prototyping may update the assumptions upon which the requirements were predicated, resulting in a change to the requirements artifacts (which would then ripple through succeeding phases).

Lack of Large Project Experience: In order to assess the applicability of a model to a project, it is often useful to compare the results of similar projects using the same model. Unfortunately, the opportunity to assess the full model has not been presented, due to the fact that the projects using the model are long-term projects that are still in the early phases. The current use of the model by complex projects with long lifespans will increase the knowledge about the model. As information from those projects accumulates, it is hoped that improvements to the model will become evident.

Need for Appropriate Automation: The fallibility of manual processes presents a major risk to systems developed using the CLLCM. Semantic modeling (in an EA/RA form) should be improved to provide the capability to precisely represent processes. Precise process representation would be the basis for the generation of automated tools to perform life cycle processes. The lack of commercial off-the-shelf (COTS) software products to support the CLLCM raises two issues. First, the lack of COTS and industry standard interfaces across the life cycle necessitates the use of manual processes and custom extensions in some areas. Second, the differences between the principles and concepts upon which the CLLCM is based and those which serve (or fail to serve) as the basis for most current COTS software products and interface models (e.g., Portable Operating System Interface for Computer Environments [POSIX, 1988]), highlight the unique perspective of, and extensive support inherent in, the CLLCM.

Several COTS products and interfaces are currently amenable to tailoring. The issue that arises when adapting products and interfaces is that others may not adapt a product or interface in the same way, limiting commonality and its related benefits. The Hierarchical Object-Oriented Design (HOOD) method, which is being used by the European Space Agency, shows promise as a basis for adaptation of a design method [CLAD, 1990]. As stated earlier, the CAIS-A is being used (and extended) to provide a standard interface. The current plan for evolution of the CAIS model consists of merging it with the Portable Common Tool Environment (PCTE) effort, resulting in a Portable Common Interface Set (PCIS) [PCIS, 1990]. In terms of tools and interfaces specifically conforming to the requirements of the CLLCM, it is hoped that the NIST standard reference model will have a clean mapping to the CLLCM and will create the necessary impetus for vendors to create COTS tools which automate the processes supported by the model. In an analogous manner it is hoped that interface standards that support the NIST life cycle reference model will emerge.

The reliability, integration, and extensibility requirements of the CLLCM go well beyond those addressed by most current COTS software products. The investment in evaluating, procuring, and training for current products may be of little use in the context of the Clear Lake Life Cycle Model. In order to minimize the risks associated with the loss of information generated using current COTS products, the model requires a means for transitioning from other models supporting various notations.

Need for Sophisticated Object and Library Management Capabilities: Essential to the implementation of the model is the ability to manage individual objects and configurations of objects. The object and library management requirements of the CLLCM are based upon the different needs and roles of technical and management team members. A discipline must be imposed on the development, maintenance, and retirement of objects as well as on the integration of objects into subsystems which, in turn, are integrated to form systems. A means must also be provided to facilitate the selection and classification of reusable objects. Unfortunately, no commercial tools have been developed to comprehensively address these requirements. Potential candidates which provide an appropriate basis for understanding and evolving object management concepts include Rational's development environment and Honeywell's Gaia [Vines, 1988]. Semantic modeling in entity attribute/relationship attribute (EA/RA) form is effective in representing important information pertaining to objects and the roles objects play within systems and reuse taxonomies. EA/RA semantic models are equally effective in representing the macroscopic issues of systems and system life cycle support environments, as well as the microscopic issues of internal object structures. The American National Standards Institute (ANSI) and the Federal Information Processing System (FIPS) Information Resources Dictionary System (IRDS) standard provide capabilities for EA/RA modeling. At the current time, there are still several other organizations supporting significant changes to the IRDS. For example, the International Standards Organization (ISO) proposal removes the attributes from relationships, which would render it significantly less useful. The ability to create and manage semantic models in EA/RA form is critical to achieving the maximum potential of the CLLCM. The ANSI/FIPS IRDS standard provides a good starting point.

A library management system, with the support of an object management system, serves as a repository of systems, tools and resources and the semantic models describing the structure of these systems, tools and resources, along with their interfaces. Users working in a system life cycle support environment will require systems, tools and resources in order to perform their duties. A means must be provided to keep a user with a specific role from accessing those aspects of a system, tool, or resource that are inappropriate for the role. A library management system supporting the concept of views is needed to provide support in an automated manner. Library management of objects and configurations with fine-grained representations has not been available in commercial products (at least not described in publications of which the authors are aware). One currently available prototype product, supporting a fine-grained object representation, is the Software Life Cycle Support Environment (SLCSE) [Strelich, 1990].

By continuing to assess, plan for, and manage the risks associated with the model, it is hoped that known risks can be addressed and minimized in future refinements. Through the combined efforts of various projects, it is hoped that the leading edges of technology can be merged into solutions appropriate to support the kinds of projects that are the focus of the CLLCM.

6.0 Benefits of the Model

The benefits of a common life cycle model, whether the standard is among divisions of a corporation, contractors for an agency or corporations within an industry, are obvious. They derive from use and support, which are outside the scope of this paper. The potential impact of wide acceptance of a standard life cycle reference model based on the CLLCM will not be discussed here, as it is also beyond the scope of the paper to mandate a standard or enforce its acceptance. The benefits that will be discussed include the ability of the model to adjust to changes in technology, the integration of all life cycle activities, and the incorporation of integration and change control concerns into the process.

6.1 Ability to Adjust to Changes in Technology

The CLLCM has been designed not only to be tailorable to the wide variety of system developments being undertaken at the current time but to address the rapidly changing nature of technology and methodologies. The facility with which the CLLCM handles change derives from its use of powerful expressive models, described by EA/RA notation and developed with an object-based discipline. Another principle that the CLLCM uses to extend the current model into the future is through the leveraging of extensible interfaces (see footnote on stable interface sets). Whereas much current standardization is based on taking a common denominator of popular tools (i.e., what is available) or interfaces (e.g., POSIX), the CLLCM is based on the actual requirements of large projects. The CLLCM can also be scaled down to accommodate small projects. A key to accommodating change is to recognize the necessity for change within the projects as they proceed through the life cycle. To do this, the CLLCM addresses the "non-functional" requirements for systems, such as flexibility, extensibility, and maintainability, as well as the functional requirements for the system. The extensibility of the CLLCM acknowledges the need to change in a planned and well-managed manner. The CLLCM attempts to be "the right tool for the job" as well as the right tool for future projects.

6.2 Integrated Approach Across the Life Cycle

The use of precise models based on well-founded principles and concepts facilitates accommodation of significantly more descriptive information capture. The semantic models can be used not only to describe the design choices made on the projects supported by the life cycle but also the choices that were rejected (along with the rationale). The capture of additional information may enable later introduction of methods or tools that may not have been practical (or in existence) at an earlier point in the life cycle. The use of baselines in the object base allows the phases and activities to share common information rather than replicate information to accommodate different formats. The ability to share information increases the total information available to each activity and reduces version skew problems inherent in systems that maintain redundant data. It also reduces the errors inherent in manual re-entry of data.

The CLLCM approach is different from many current approaches that attempt to "cobble together" non-integrated COTS products (that are usually not based on precise models) and then attempt to format data for use by other COTS products. Rather than developing an environment in which COTS products may be rendered obsolete by changes to a method (or by introduction of a new or additional method), the semantic models of the CLLCM provide the opportunity to generate a new instance of a tool. The generated tool can be amenable not only to the existing information (which has been captured in the project baseline) but also to the new information particular to the changed (new or added) method. In this way, system developers are never faced with the choice of continuing with less than adequate tools or enduring a significant discontinuity in productivity as current artifacts are brought up to the new method.

6.3 Process Incorporates Integration and Change Control

All activities of system development are addressed in all phases of the CLLCM. This is different from most life cycle processes that deal directly with development but do not specify the points at which software quality assurance, change management, project management or integration enter into the process. The framework concept inherent in the CLLCM supports the life cycle management of integration and change control. Figure 4-1 shows the interactions of the project and configuration management and the quality assurance organization with every artifact produced throughout the life cycle. Because the semantic models of the object base describe processes and interfaces as well as artifacts, tailored models of the first and second levels of abstraction (Figures 4-1 and 4-2) will be an important part of each project's baseline. Specifically, these models will be used for active control of the manual and automated processes throughout the life cycle.

The seemingly contrary nature of the concepts of integration and change control present a significant challenge to development of a cohesive model. Whereas integration strives to foster similarity, change control manages dissimilarities. Not only must the model be able to integrate artifacts (in potentially different forms than originally envisioned) as the life cycle proceeds but it must also be able to manage the various changes (e.g., configurations, requirements, and staff) that occur throughout the life cycle for a system. Management of objects in the object base according to baselines in each phase eases integration. When a new baseline is introduced, it can be done in a manner that allows a period of changeover (i.e., users can study the implications of the proposed baseline and plan accordingly) as well as the possibility of regressing back to the previous baseline (if problems arise after the new baseline becomes operational). This capability is a critical need of large projects developed on an incremental basis. Change management is facilitated by capturing all information relevant to each release within the object base. By using the baseline to manage the increments of integration and the information relevant to change management, the two activities are reconciled in a cooperative manner.

The concept of stability (as in stable interface sets) also aids integration and change management. As stated in Section 4.3, the semantic models associated with stable frameworks allow the impact of integration and change to be assessed before an actual change takes place. In this way, areas of concern can be revealed and scheduled for additional effort (which might be in the form of prototyping) in order to smooth the process of integration. The concept of stability (i.e., systematic and documented methods for extending the capabilities or responsibilities within the provided perspectives) provides a well-ordered migration path. It is hoped that the benefits of the model (and its improvements) will lead to wide acceptance and support by the systems engineering community. It is intended that the model be simple enough to reason about but that it support enough complexity to handle current and future system developments. Success will be measured in terms of the number of implementations of the Clear Lake Life Cycle Model that can meet those challenges.

7.0 Concluding Remarks

The dynamic and unpredictable nature of system technology and methodology advances has resulted in the need to handle increasing complexity when dealing with system development projects. When the project complexity is coupled with the complexity of the systems to be developed, *ad hoc* methods will almost certainly fail. Precise models that can be described in a semantically unambiguous manner are needed to describe the system, the life cycle environment, and the interactions between the system and its life cycle.

The requirements for software systems, especially in the domains of defense, aviation, and space applications, are becoming more critical. Software contractors are being asked to deliver systems that are large, distributed, real-time, continuous (experiencing little, if any, down time), and able to support extensibility and tolerate faults. However, life cycle models have typically applied to small and simple systems. Many such models cannot be scaled up to handle more complicated applications. On the other hand, it is easier to scale down a software engineering life cycle model geared toward large, complex systems to meet the requirements of simpler applications. Therefore, life cycle models intended to help resolve the "software crisis" must address the high end of software complexity by leveraging advancements in theoretical foundations as well as the corresponding technologies.

The activities of the life cycle, from concept exploration through maintenance and operation to retirement, provide a framework that incorporates software engineering at each step. The Clear Lake Life Cycle Model is also amenable to various organizations, although an iterative process may be the most compatible with the principles of software engineering. The CLLCM is an emerging life cycle model that is being evaluated for use by a number of organizations and will benefit from the use and by the findings of those organizations. The use of well-engineered, emerging standards is intended to leverage expertise of other groups, in order to speed acceptance and automation of the CLLCM. The risk-directed focus on substantial, complicated systems, developed in an incremental manner over a protracted period of time, attempts to tackle the most difficult system requirements. The attention to integration and change management distinguishes the CLLCM from other models. By addressing the most difficult systems using a tailorable strategy, CLLCM can also be used for less complex systems. The life cycle approach is intended to be explored, evolved, and refined to address the life cycle issues of an incrementally evolving project, incorporating new technologies and new methodologies over a significant period of time.

Acknowledgements

The Clear Lake Life Cycle Model for System Engineering was created, communicated, and motivated by teams led by Dr. Charles McKay, Director of the NASA Software Engineering Research Center and the High Technologies Laboratory at the University of Houston - Clear Lake. The authors of this document acknowledge the participation of many researchers in the refinement of this model. Special thanks are due to Charlotte Wilford whose editing substantially improved the readability of this paper.

All errors of commission and omission are solely the responsibility of the authors.

References

[Booch, 1987] Booch, G., *Software Components with Ada*, Benjamin/Cummings Publishing Company Inc., Menlo Park, California, 1987.

[Boehm, 1988] Boehm, Barry, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, May 1988, pp. 61-72.

[Boehm, 1989] Boehm, B., *Software Risk Management*, IEEE Computer Society Press, 1989.

[Burns, 1989] Burns, A. and C. McKay, "A Portable Common Execution Environment for Ada", *Ada: The Design Choice - Proceedings of the Ada-Europe International Conference, Madrid, 1989*, Cambridge University Press, 1989.

[HOOD, 1989] HOOD Reference Manual, Issue 3.0, WME/89-173/JB, September 1989.

[POSIX, 1988] IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std 1003.1-1988, Institute of Electrical and Electronics Engineers, 30 September 1988.

[LMSC, 1989] Lockheed Missiles and Space Corporation (LMSC) NASA Software Support Environment (SSE) DRLI 58, SSE Architecture Design Document, November 1989.

[McKay, 1988] McKay, Charles W., "Conceptual and Implementation Models

Which Support Life Cycle Reusability of Processes and Products in Computer Systems and Software Engineering", RICIS Research Report for AIRMICS, AIRMICS-HTL Grant 2-5-51537, 1988.

[CAIS-A, 1989] Military Standard Common APSE Interface Set (CAIS) DOD-STD-1838A, 6 April 1989.

[McKay, 1989] McKay, Charles W., "Some Notes on Risk Management", 1989.

[Muncaster-Jewell, 1988] Muncaster-Jewell, Penny, "Change Management Needs for Persistent Data Bases (PDDs)", *Proceedings of the Third Annual Knowledge Based Software Assistant (KBSA) Conference*, August 1988.

[Parnas, 1972] Parnas, David L., "On the Criteria To be Used in Decomposing Systems into Modules", *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.

[CLAD, 1990] Reference Manual for the Clear Lake Approach to Design (CLAD), Version 1.0, Software Engineering Research Center at the University of Houston at Clear Lake, 22 January 1990.

[Royce, 1970] Royce, W. "Managing the Development of Large Software Systems: Concepts and Techniques", Proceedings of Wescon, August 1970.

[PCIS, 1990] Solomond, J., "Letter from the Director, Ada Joint Program Office", *Ada Information Clearinghouse Newsletter*, Vol. VIII, No. 1, March 1990, p. 1.

[Strelich, 1990] Strelich, T., "Software Life Cycle Support Environment", General Research Corporation, RADC-TR-89-385, February, 1990.

[Vines, 1988] Vines, D. and T. King, "Gaia: An Object-Oriented Framework for an Ada Environment", *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, May 1988, pp. 81-90.

About the Authors

Kathy Rogers is a Member of the Technical Staff at the MITRE Corporation in Houston, Texas. She has over eight years of experience developing software systems. Her recent responsibilities have been in the evaluation of Ada and software engineering methods, processes, and products for the NASA Space Station Freedom program. She received a B. S. in Computer Science and a B. A. in Economics from the University of California at Irvine in 1982. She is currently working toward an M. S. in Computer Science at the University of Houston at Clear Lake. Ms. Rogers is a member of the IEEE, ACM, and National Special Interest Group in Ada (SIGAda) and was the Chair of the Clear Lake Area (Houston) chapter of the SIGAda in 1988 and 1990.

Michael Bishop received the B.S. degree in computer science from the University of Houston in 1984. He is currently working toward the M.S. degree in computer science at the University of Houston at Clear Lake with a research emphasis in semantic modeling. Mr. Bishop has over six years of experience as a software engineer working in aerospace applications. He is currently a Senior Software Engineer at Unisys. Mr. Bishop is a member of the Association of Computing Machinery, the IEEE Computer Society, and the National SIGAda.

Dr. Charles W. McKay is the founding director of the NASA-chartered Software Engineering Research Center (SERC) and the Texas Higher Education and University of Houston-chartered High Technologies Laboratory (HTL). The SERC provides focused research to advance technology in computer systems and software. Dr. McKay is the Team Leader and Principal Investigator on the Portable Common Execution Environment (PCEE) project, the University of Houston at Clear Lake (UHCL) Technical Director of the AdaNET project, a Technical Advisor for the Boeing STARS team, and Chair of the Ada Runtime Environment Working Group (ARTEWG) subgroup responsible for the Catalog of Interface Features and Options (CIFO). Dr. McKay has more than 20 years of experience in research, development, and teaching of computer automated systems. He is the author of three textbooks, numerous articles and reports, and numerous video taped lectures and courses.

QUALITY ASSURANCE REQUIREMENTS FOR AN EVOLUTIONARY DEVELOPMENT METHODOLOGY

Richard M. Lobsitz, Peter G. Clark, and C. Robert String

TASC
Reading, Massachusetts

Abstract: Evolutionary development models offer greater flexibility in setting system requirements than standard waterfall models. However, without innovative quality assurance practices, the software development process can degrade to an uncontrolled, poorly tested state. This paper discusses issues unique to evolutionary models in the areas of requirements baselines, configuration management, and certifying operational readiness. A quality assurance program to support our TASC-EDGE™ (Effective Development through Growth and Evolution) development methodology addresses these issues. The model is designed to manage the risk associated with poorly defined requirements or requirements that emerge from the use of fielded systems rather than exist a priori. This paper describes the quality assurance guidelines which define a complete set of reviews, documentation, and testing milestones for the TASC-EDGE software development life cycle. These guidelines structure the evolutionary development so that the system is allowed to evolve with control over the scope and timing of changes to the requirements. The TASC-EDGE life cycle recognizes that quality assurance is based on a combination of verification activities that demonstrate system conformance to specifications, and validation/evaluation activities that assure that the system meets users' real requirements.

INTRODUCTION

The software industry has defined various models of development life cycles. Two models have gained notable popularity: the classic "stepwise refinement" life cycle (often referred to as the "waterfall model") and the evolutionary or incremental release model. *Both models strive to develop a usable software product, but each has a different quality assurance (QA) process that affects the final quality of the software.* They begin with different assumptions, have different goals, and yield different results.

This paper first reviews the QA assumptions, goals, and results of the waterfall model. After briefly describing the evolutionary software development model, the QA assumptions, goals, and results of the evolutionary model are discussed. Next, we describe some of the issues that are unique to the evolutionary development model. The paper goes on to discuss all of the phases in the TASC-EDGE life cycle and the associated quality assurance activities. We conclude with a brief summary.

Quality Assurance in the Waterfall Model

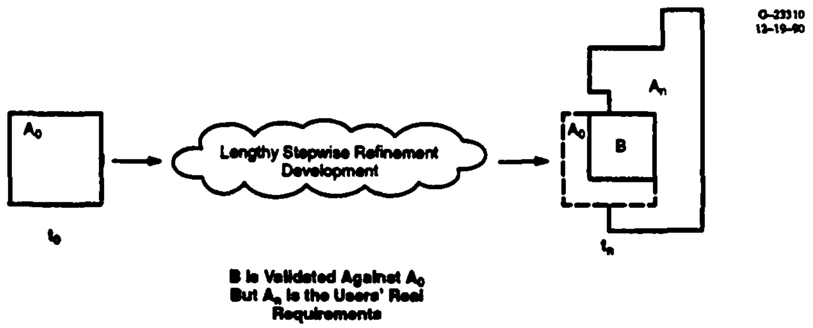
The waterfall model of software development proceeds by means of an orderly sequence of transformations from requirements to design to code, in linear order. The software system being built is defined in increasingly greater detail as it progresses through the model's life cycle phases. Advancement to the next phase is dependent on finalizing (freezing) and approving the results of the previous phase. The final product is software that implements the requirements defined and frozen in the earliest phases.

In the waterfall model, the fundamental assumption is the requirements "as-specified" at the end of the requirements analysis phase are nearly identical to the users' real requirements. As a consequence of this assumption, the goal of the QA program is to verify and validate (V&V) the system under test against the as-specified requirements. The system testing is designed to identify any "shortfall" between the system under test and the as-specified requirements and, to a lesser extent, the "gold plate" where the system under test exceeds the as-specified requirements. The QA process in the waterfall model emphasizes closure of each phase and a sense of finality to the end-product; the system is declared "finished."

Figure 1 shows what happens during system testing (t_n) when the users' real requirements (A_n) are not identical to the as-specified requirements (A_0). The reasons for this requirements mismatch are discussed by Brooks (Brooks, 1987) and others. However, in the waterfall model, there is strong resistance by all parties against restating requirements for a number of reasons. First, it is not the V&V organization's job to restate requirements. Their job is to point out differences between the system under test (B) and the as-specified requirements. Second, the developers usually don't know enough to determine whether the system under test should be modified or a requirement should be changed and, since changing a requirement may have far-reaching consequences, they opt to modify the system under test. Finally, the buyers, and possibly users, don't want to appear foolish by now changing the requirements that they had previously accepted.

When the real users finally get the system it is apparent that the true gold plate is different from the gold plate identified by the QA process. It may, in fact, extend into areas that were previously thought to be shortfalls. In addition, there is a real shortfall in the areas not addressed by either the as-specified requirements or the system under test. Furthermore, the situation rapidly deteriorates as the mismatch between the as-specified requirements and the users' real requirements increases.

™ TASC-EDGE is a trademark of The Analytic Sciences Corporation



G-23310
12-19-90
A$_0$
Lengthy Stepwise Refinement Development
A$_n$
A$_0$
B
t$_0$
t$_n$
B is Validated Against A$_0$
But A$_n$ is the Users' Real Requirements

Figure 1 Typical Waterfall Model QA Results

The net result is that the developers spend valuable time and money after test making the system meet as-specified requirements that may not be real, while totally ignoring unspecified areas that may be critical to the real users. *Nobody has really learned anything about the users' real requirements from the process, nor have they been required to do so by the process.*

The Evolutionary Model

TASC's Effective Development through Growth and Evolution (TASC-EDGE) model addresses the requirements uncertainty problem by using a "natural growth" model which starts with a high level view of the eventual product and extends and refines its capabilities at regular intervals in response to user feedback on emerging deployed capability until the final product is achieved. This is accomplished by combining classical life cycle activities into iterative enhancement at the development level. TASC-EDGE has evolved from our extensive experience in developing large information systems for our Government customers. It began as an attempt to effectively address the observed weaknesses of software development standards such as DoD-STD-7935A, DoD-STD-2167A, and DoD-STD-2168. It also uses ideas adapted from the work at the Software Engineering Institute (SEI) on software development process models and other advanced software concepts. It is a combination and extension of Boehm's spiral model (Boehm, 1986) and Martin's Time-Box development approach (Martin, J., 1986). It incorporates structured and prototype-based development techniques to achieve maximum customer satisfaction within the constraints of the project schedule and budget. These techniques promote greater flexibility to cope with the dynamic nature of advancing technology and the continual change in most end-users' environments and roles within their own organizations.

TASC-EDGE sets up a top-level evolutionary plan and produces the system in several increments, allowing feedback from the users to direct the development. The basic design philosophy behind evolving a system at TASC is to layer complexity into the system. This means starting with a simple implementation of a requirement and adding complexity in subsequent increments after the basic capability has been completed. The methodology is constructed so that the competing factors of schedule, budget, and system functionality can be jointly managed to deliver scheduled incremental capabilities. (Lobsitz, 1988)

Figure 2 is an overview of the steps in the TASC-EDGE model showing multiple cycles for developing a software system. Each cycle begins with the **Systems Concept and Definition** activity. Here, the users and developers (re)baseline the system requirements and (re)set implementation priorities. This activity should be of relatively short duration and detailed analysis should be confined to what is being planned for the next increment. Thereafter, requirements are drawn out of the users by letting them evaluate already delivered capabilities. Once an increment baseline is established that reflects the priorities of the users and buyer, a **Time-Box Development** activity begins. The time-box concept establishes a fixed schedule and budget for the developers, but allows flexibility in the content of the delivered product (on the condition that the high priority functions are completed first). When the time-box development is finished, any unimplemented, low priority functions are reprioritized in the next cycle.

System Operation allows the users to experience the system in action and the developers to collect statistics and **User Feedback**. System performance is monitored and tracked for evaluation. Finally, requested changes, usage statistics, performance statistics, specifications of deferred functions, and new system requirements are collected and analyzed to establish a new baseline (agreed to by all parties) for the next development cycle. (Clark, 1989)

Prototypes are used throughout the process to support the definition of the system concept, to elicit and refine detailed requirements, and to do performance estimation. These prototypes aid in demonstrating the interfaces of the system to related, external systems, identifying high level functionality and response time requirements for the system, and gauging data volume and throughput. This is often done by simulating the user interface of the system to identify not only user interface requirements, but functional and performance requirements as well. Operational prototypes are the building blocks on which the system is grown. Here we quickly build the increment by integrating commercial, off-the-shelf (COTS) tools (e.g., database management systems or spreadsheet) and reusable components with custom-built 4GL and 3GL modules which are either automatically generated from front-end computer aided software engineering (CASE) tools or coded and tested by hand using current language-sensitive editors, compilers, and debuggers.

One of the major advantages of evolutionary development is its emphasis on user involvement and the feeling users have that they "own" the system because their input is solicited and acted upon. For the evolutionary approach to be effective,

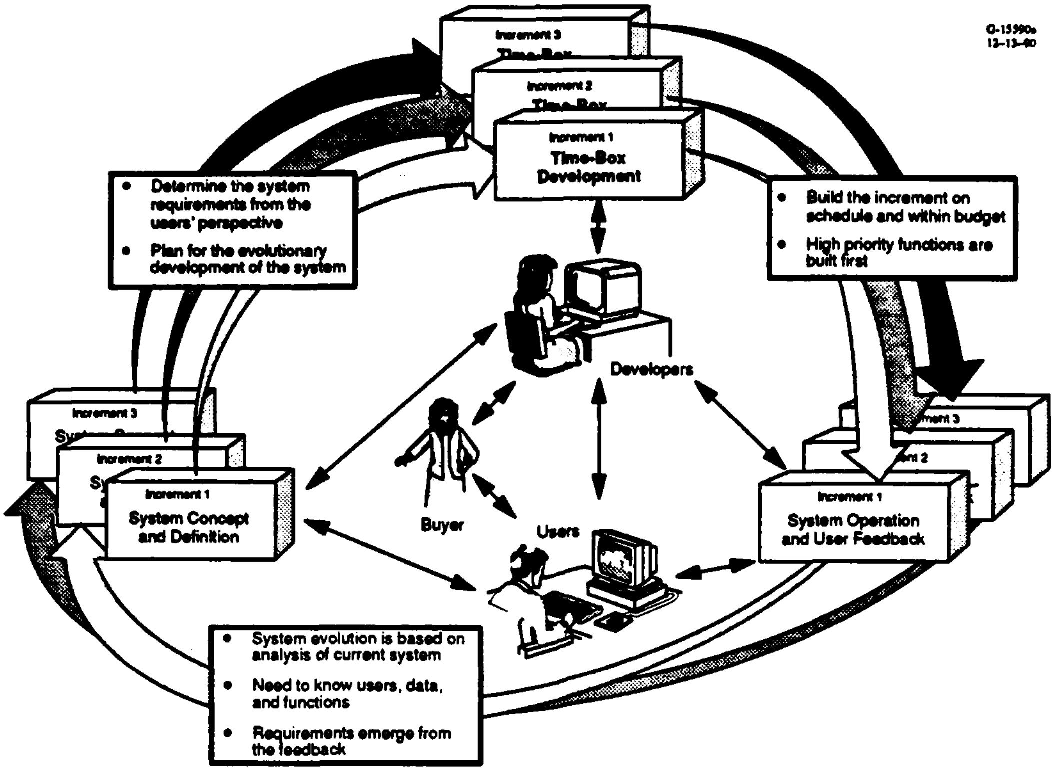

G-15390a
12-13-90
Increment 3
Increment 2
Increment 1
Time-Box Development
• Determine the system requirements from the users' perspective
• Plan for the evolutionary development of the system
• Build the increment on schedule and within budget
• High priority functions are built first
Developers
Buyer
Users
Increment 3
Increment 2
Increment 1
System Concept and Definition
Increment 3
Increment 2
Increment 1
System Operation and User Feedback
• System evolution is based on analysis of current system
• Need to know users, data, and functions
• Requirements emerge from the feedback

Figure 2 Steps in the TASC-EDGE Model

feedback must be collected from the user. If the users cannot make timely comments or feel that their inputs are going unheard, they will stop providing the feedback and eventually stop using the system. Techniques used here can be manual, such as providing "hot-line" support or user group meetings; automated in a passive manner, such as providing an integrated, on-line change request function perhaps connected by E-mail to the development team; or automated in an active way, by having a performance monitor embedded into the system which logs user activities and system responses. The feedback data must then be analyzed by the developers and users to provide guidance to the next evolutionary build.

Quality Assurance in the Evolutionary Model

In an evolutionary model, the underlying premise of the QA process is that the requirements as-specified at the end of each incremental requirements analysis phase are incomplete and lacking in detail. Thus, the goal of the QA program is to use operational software that implements the high level requirements to identify and clarify the users' detailed requirements. The system testing is designed to verify and validate that the high priority as-specified requirements in the "as-built" increment work correctly and to ensure the reasonableness of the unspecified requirements that are implemented as a result of the design and coding process. The harder QA work takes place after putting the system into operational use. There, the QA process must be designed to evaluate the users' feedback and other data to separate the real requirements from the incorrect ones. The QA process in the evolutionary model seeks to find opportunities for refinement and enhancement primarily through the evaluation of user feedback and system performance statistics. The system is never really "finished" in that all system requirements are satisfied (this is not achievable due to the dynamic nature of the user environment), rather, all of the highest priority requirements have been implemented to the satisfaction of the user community as a whole.

Figure 3 shows what happens at various times (t_i) throughout the evolutionary development when the users' real requirements (A_n) are not identical to the as-specified requirements (A_0). In the evolutionary model, the bias is strongly in favor of restating and refining requirements. At the end of each incremental cycle, the operational software (B_{i-1}) and performance data serve to define new and more detailed requirements (A_i) via feedback from the users. This process continues for the duration of the software acquisition. In this case, the final system (B_{n-1}) matches closely to the requirements as they are known at the time (A_n).

There are several situations that occur in this process which help to define requirements. Two are the result of the shortfall between the as-built increment and the as-specified increment and two are the result of gold plating or implementing unspecified requirements in the as-built increment. The first situation, and perhaps most serious, is that of "real shortfall." Here, low priority requirements were dropped from the increment since the developers ran out of time and/or money. But if the shortfall is real, the users will raise the priority of those requirements so that they are implemented in a subsequent increment. The second situation is called "fortuitous shortfall"

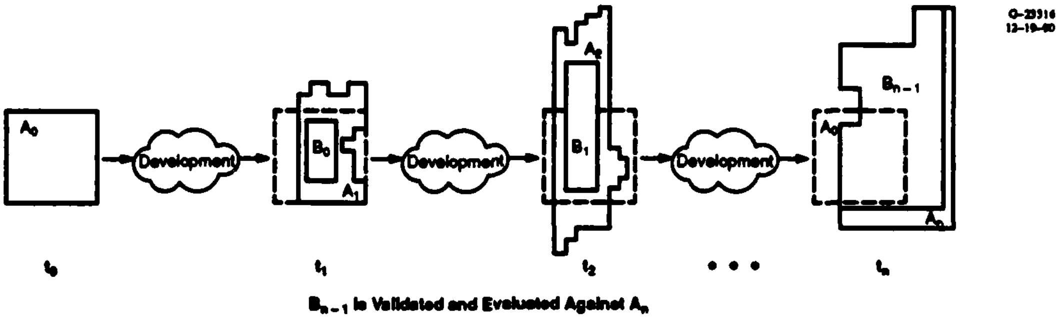


Figure 3 Typical Evolutionary Model QA Results

because it covers the low priority requirements that were dropped from the increment and it turns out that the users, after using the operational software, decide they are unnecessary or continue to be low priority. The next situation is called "false gold plate" where the as-built increment implements unspecified requirements that turn out to be real. Finally, there is "true gold plate" when the as-built increment implements either specified or unspecified requirements that turn out to be unneeded or even wrong. This situation is often benign to the majority of the users, but of significance to the buyers and developers because it consumes resources and may unduly constrain subsequent development.

The magnitude of these situations will vary, but all four situations are likely to arise. Moreover, *notice that even though some of the situations may cause problems for the users, all of the situations serve to identify and validate the users' real requirements* and that all of the problems can be fixed in later increments.

QUALITY ASSURANCE ISSUES IN EVOLUTIONARY DEVELOPMENT

To this point, we have discussed the contrasting goals of the evolutionary vs. classic waterfall life cycle models. Both models have strengths and weaknesses. When applying evolutionary development to critical, ultra-reliable systems (especially Mission Critical applications), the model must address issues that could negatively affect the quality of the delivered product. The TASC-EDGE life cycle model is a variation of an evolutionary model — it offers solutions to issues in three life cycle areas:

- Establishing a requirements baseline for developers while actively soliciting new requirements
- Applying configuration management techniques to evolving subsystems on different development tracks
- Certifying that a system release is ready for an operational environment.

In the following discussion, we will explore the impact of each issue on quality assurance practices. Solutions offered by the TASC-EDGE model will be described.

Establishing A Requirements Baseline

Fundamental to the evolutionary development model are incremental "design/code/test" cycles where the requirements for each release are refinements from previous specifications, or newly defined requirements. Both types of requirements can be derived from users' feedback on existing (delivered) releases. To maximize the inclusion of new and modified requirements in an upcoming development increment, users' feedback is typically solicited prior to the design of the next increment.

During design and implementation, developers must aim for a steady target — a requirements baseline — to ensure that the next release will be both verifiable and valid. Classic waterfall development models foster this by "freezing" requirements at a preselected point prior to any design work. Evolutionary development models, however, continue to collect new or modified requirements as development progresses. At what point in an evolutionary development life cycle should the requirements be frozen and a baseline established? When is it acceptable to "thaw" them, revise specifications, and establish a new baseline without impacting consistency, quality, or validity?

TASC-EDGE Baselines for Freezing Requirements — At the beginning phases of the life cycle, the TASC-EDGE model defines two baselines — the Functional Baseline and the Increment Baseline. These baselines are important in solidifying requirements prior to design and implementation work, but also provide vehicles for modifying requirements to allow for system evolution.

The successful completion of a System Requirements Review (SRR) establishes the Functional Baseline. This baseline captures all known high-level requirements to be met by the overall system. The requirements are tagged with a relative priority indicator and are documented in the High-Level Requirements Specification. The Increment Baseline is established following the successful completion of an Increment Baseline Review. This baseline constitutes a subset of total system requirements — predominately those requirements that currently have the highest priorities.

At this point in the life cycle, design, implementation, verification, integration, and system validation can build towards a static target — the Increment Baseline. Each new or modified requirement (and its accompanying priority) received from concurrently fielded system releases are added to the High-Level Requirements Specification with a "pending" status. These new requirements are not yet part of any baseline, but eventually may be.

Reestablishing Baselines — After a developed increment is integrated into a system release, results from operational testing or user feedback on the installed release may be justification for revisiting High-Level Requirements. This invokes the next iteration of the evolving system. All requirements and priorities gathered during development and post-development activities are reevaluated and substantiated in another SRR, resulting in a new Functional Baseline. In effect, the requirements have "thawed" and another Increment Baseline is established for developers. The new baseline reflects users' experience, new knowledge, and deeper understanding about actual system requirements.

Validation of a completed system release is accomplished by functional testing. Testing criteria work well for low-level requirements, but a system's satisfaction of high-level, often intangible requirements (e.g., usability) requires an additional validation component. TASC-EDGE employs system evaluation as a necessary step in certifying a release. Evaluation is a mixture of two primary areas: users' calibration of subjective validation criteria and improvement recommendations. Evaluation feedback drives the next increment, reestablishing the next series of baselines.

Separately Evolving Configuration Items

Successful development of a large, complex system hinges on many technical and management elements. A fundamental element is partitioning the development of the total system configuration into a group of more easily managed subsystems. Proper communication and interaction among integrated subsystems are crucial to the quality of the finished product. Classic waterfall development models promote this by managing single (and typically parallel) development efforts for each subsystem. Each subsystem is managed as a configuration item. Careful configuration management (CM) of requirements, designs, code, interfaces, and documentation ensures that each subsystem stays "current" with respect to the other system configuration items.

When using an evolutionary development model, CM is a greater challenge. A large, complex system may still be partitioned into subsystems, but each subsystem may iterate and evolve seemingly independent of other parallel subsystem development efforts. As each subsystem evolution "goes its own way," how can system-level management assure the quality and integration success of a planned system release? How can CM techniques prevent separately evolving configuration items from jeopardizing interfaces?

Multiple CM Levels — Consistent with sound design principles emphasizing information hiding (Parnas, 1972) and functional independence, any large system should be partitioned into relatively isolated subsystem development efforts regardless of the life cycle model being followed. The TASC-EDGE model defines multiple levels of CM responsibilities, depending on the size and complexity of the total system.

For example, a system comprised of a lower-level collection of subsystems could employ two CM activities. The top (system) level CM would be managed by a Steering Group. This group has primary responsibility and control over the relative priorities of the requirements and determines, for the most part, the contents of Incremental Baselines. The steering group also provides guidance that often influences the "direction" in which the total system evolves.

Lower level subsystem CM activity would be managed by more classical Configuration Control Board (CCB) groups. One CCB for each parallel subsystem development effort is needed. Working within the constraints of the Increment Baseline, these lower level CM groups manage the identification, status accounting, change control, and auditing requirements (MIL-STD-483A, 1985) of the developing subsystems.

Boundaries Between CM Levels — Within the TASC-EDGE model, the critical boundary of responsibility between any two CM levels is defined by interface specifications. Changes to interface specifications that potentially or realistically impact upper level integration must be reviewed and approved by the next higher CCB level.

The importance of interfaces is emphasized in the TASC-EDGE model by requiring all interface definitions to be frozen early in any increment development. Interfaces must be defined no later than the System-Level Design phase, and are approved at the System Design Review. Beyond this point, subsystems may iterate and evolve independent of other parallel efforts, providing interface integrity is maintained.

Certifying Readiness for an Operational Environment

Classic waterfall models assert development frameworks relatively unsympathetic to users' changing requirements — the models advocate a path of stepwise refinement that originates from an unyielding (and often painfully established) requirements baseline that is assumed correct. While increasing the risk of delivering an obsolete system, the waterfall model minimizes errors and inconsistencies in the development process by sacrificing requirements flexibility. Quality assurance work, namely verification and validation, can be applied rigorously. Consistency and completeness of work is certified before advancing to the next step; final system validation is performed by comparing the as-built system with the original requirements baseline.

Evolutionary development models provide a different result: they minimize the risk of delivering an obsolete system, but typically are less structured in development process formalities. If left unchecked, validation and certification of a system's operational readiness (especially for Mission Critical applications) can be an issue. How does the certification team ensure that all system components have been developed to satisfy the "correct" and most current requirements?

Tracking Defects and Enhancements — Tightly managed "discrepancy" tracking is incorporated into TASC-EDGE to assist in certifying system releases. As the overall system iterates, evolves, and approaches a release point, a discrepancy in any increment is clearly categorized into one of two categories: a maintenance order (MO) or an enhancement request (ER). Maintenance orders document defects — system behavior that deviates from the specification. Enhancement requests are additions or improvements that act as catalysts for evolution.

Prior to Operational Testing, all "open" MOs must be resolved, determined innocuous by the responsible CCB or top level steering group, or fully documented as an operational testing issue. ERs are collected and each is tagged with a priority. Each ER's disposition and potential influence on system evolution will be evaluated in the next increment's high-level requirements phase.

Defects vs. enhancement distinction, rigorous ER/MO tracking, and successful completion of operational testing are key elements in certifying that a system developed using TASC-EDGE is ready for an operational environment. Follow-up on corrective action further enhances TASC-EDGE software quality. Corrective action eliminates recurring errors by diagnosing and correcting problems that cause errors. These QA activities combine to minimize the risk of incorporating a defective development increment into any system-level release.

Incremental Releases Increase Certification Confidence — Confidence in having achieved testing thoroughness is a key criterion in a system's ability to pass certification. A life cycle model stressing incremental releases raises testing confidence at certification time because the system has already progressed through multiple, tested releases. Each release has, by definition, passed validation. The system has been "shaken out" multiple times before operational certification is attempted. Furthermore, as a system evolves using TASC-EDGE, completeness of validation test sets increases as a natural by-product of evolving requirements. Automated tools assisting in regression testing can streamline validation efforts and provide a high degree of accuracy in discovering incremental defects.

DESCRIPTION OF THE TASC-EDGE QUALITY ASSURANCE PROCESS

The TASC-EDGE development life cycle is designed to provide the greatest flexibility in accommodating changing requirements while still controlling the transformation from requirements to design to code. Figure 4 illustrates the development phases within the TASC-EDGE development life cycle and each phase is described in detail in the following sections. Table 1 defines the documents, software specification data, and reviews required for the delivery of an operational system under the TASC-EDGE development life cycle. It is beyond the scope of this paper to describe the contents of all these documents and data items; however, brief descriptions are included when the items are mentioned in the description of the development phases. The use of CASE tools and automatic document generation is strongly supported by the QA requirements to prevent any delivered documentation from being out of "synch" with the delivered software.

The table lists the documents, data, reviews, and baselines against the eleven different life cycle phases. The dark boxes indicate when a particular item or activity is considered "deliverable" either internally or to a customer, the light portion of the horizontal bars indicates that the item is being created or updated during that phase. Some of the documents and development data have multiple deliveries, indicating a preliminary version(s) and then a final version. This supports the general principle of creating an "as-built" version of the development data prior to the completion of a configuration item or the release of an increment.

As illustrated in Figure 4, several iterative loops are built into the development cycle. At the lowest level, there is a loop involving Low-Level Design, Implementation, and Verification test called the **Configuration Item** development cycle. This cycle allows for a design to be iterated on before it becomes frozen and released for integration into an increment. This loop is

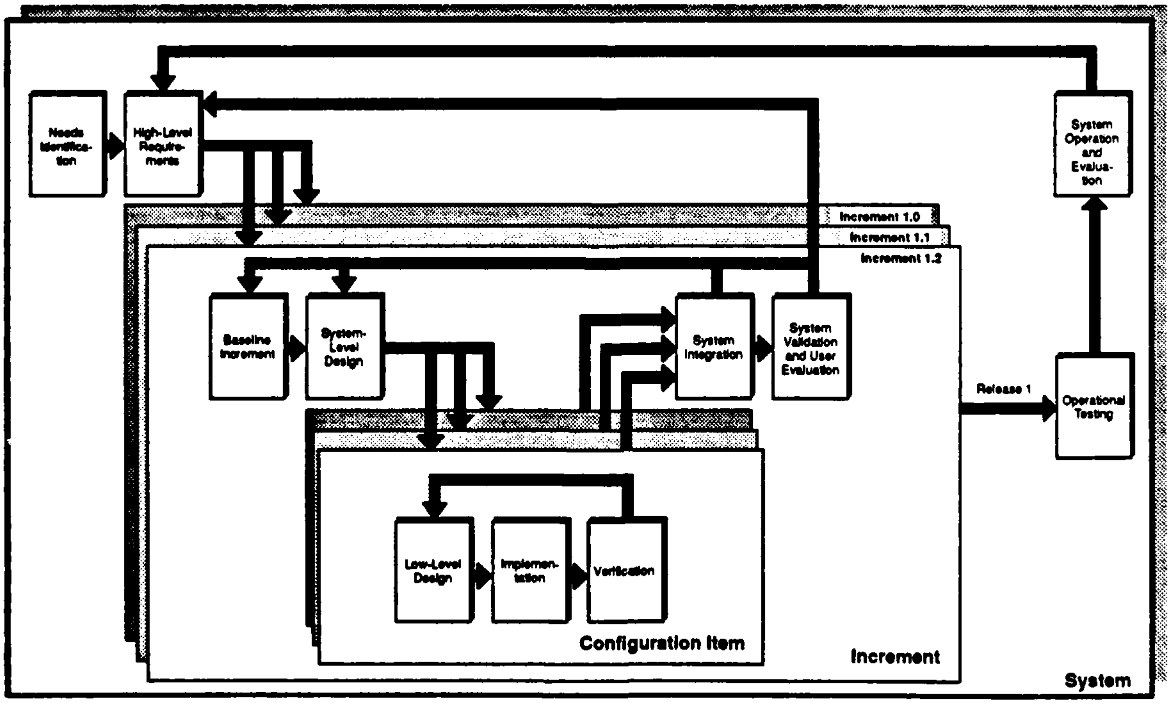


Figure 4 Iterative Phases in the TASC-EDGE Methodology

Table 1 Software Quality Assurance Requirements for TASC-EDGE

O-23257
12-31-90

TASC–EDGE Life Cycle Phases

Software Quality Assurance Requirements	Needs Identification	High–Level Requirements	Baseline Increment	System–Level Design	Low–Level Design	Implementation	Verification	System Integration	System Validation and Evaluation	Operational Testing	System Operation and Evaluation
Concept of Operations/Prototype	□	■	□								
System Development Plan		■	□	□	■						
Detailed Requirements (Functions and Interfaces)				■	□	■					
Test Plans and Specifications				■	□	■	□	■			
Design Specifications				□	■	□	■	□	■		
Release Documentation						□	■	□	■		
Maintenance Orders								□	□	□	□
Enhancement Requests		□	□	□	□	□	□	□	□	□	□
Reviews		■	■	■	■	■					
Baselines		■	■				■		■	■	

Legend:

□ Created or updated

■ "Delivered"

controlled by having to deliver the functionality specified in the System-Level Design Phase.

The next level loop is defined as the completion of a specific Increment of system capability which may incorporate multiple Configuration Items. This loop spans the Baseline Increment phase to the System Validation Phase. This loop allows the development process to evolve the system requirements by incrementally building the system and reassessing the system requirements through system validation testing. This loop is controlled by the Baseline Increment Phase which produces an increment baseline description.

The final loop is the System development cycle which can contain one or more releases that have passed through a rigorous operational testing. Changes to the system, once it has achieved an Operational Baseline, can reset the development cycle as far back as a reassessment of the High-Level Requirements, or can involve the development of another set of increments to achieve the next level of planned-for capability.

The following sections provide a brief overview of the activities and quality assurance requirements for each of the eleven development phases.

Needs Identification — The purpose of this phase is to determine and evaluate a potential project's scope of effort, level of organizational support, assessment of risks and likelihood of success. Development successes/failures of other similar projects are evaluated to incorporate their "lessons learned." This phase typically involves creating a proposal for the development project and gaining organizational suppon. The output of this phase is the agreement to proceed with the development project.

High-Level Requirements — This phase identifies the scope and relative priority of the system's required functions from the users' perspective, identifies system requirements from the developer's perspective, and plans the development project. This phase is ended by successfully completing a System Requirements Review and establishing the Functional Baseline. The focus of this phase's activities is the creation of a Concept of Operations that describes to the users what the new system will do and how it will perform. Typically, this involves prototyping important system functions and user-centered requirements analysis (Martin, C.F., 1988). Once the high-level requirements are understood, detailed requirements are deferred to the System-Level Design Phase.

The Concept of Operations includes implementation priorities for each described function as well as performance requirements tied to these functions. Finally, a System Development Plan is created to describe how the system will be developed incrementally based on developing the highest priority functions first and adding lower priority functions in later increments. This plan also includes describing the application of the quality assurance process and the project's configuration management practices. The Quality Assurance activities in this phase center on the review of the Development Plan and ensuring that the users are satisfied with and understand the Concept of Operations.

Baseline Increment — This is a relatively brief activity but crucial to the successful use of the Time-Box development method. This phase creates a baseline defining the next development increment. This Increment Baseline describes the subset of the high-level functions to be developed in the current increment and the relative priorities of each of those functions. During the Increment Baseline Review, schedules and budgets are set for the completion of the increment and the development priorities are firmly defined. QA is responsible for ensuring that the baseline is well defined and that all parties agree to the established priorities.

__System-Level Design__ — This phase is really a combination of requirements analysis and top-level design, it creates or expands on the detailed requirements by first identifying specific requirements based on the high-level requirements prioritized for this development increment. These detailed requirements are then assigned to the various system components and the interfaces between the components are defined. As detailed requirements are identified, matching test criteria are also identified. A detailed requirement cannot be defined if the test criteria cannot be established. The result is that each of the components is allocated a set of required functions and interface specifications are developed to define how each component interfaces to the other components in order to satisfy the detailed system requirements. A Detailed Requirements Review is held to review the results of the analysis for this increment. Configuration control is established for the interface specifications so that any changes to the specifications required by subsequent design or development activities of a particular component is communicated and agreed to by all affected components.

__Low-Level Design__ — This activity is accomplished on a component by component basis and uses the Detailed Requirements as the design guidelines. The results of this design activity create a preliminary version of the Design Specifications for a component which is reviewed at a Software Design Review. The draft version of the component design specifications will not be completed until the Verification Phase, at which time, an audit of the implemented software will update the design information with the actual implemented design. This allows small changes to the design to be incorporated without a large configuration control overhead; however, changes to the component interfaces have to be reviewed at a higher level. The QA responsibility in this phase is to ensure that the design specifications are reviewed by a panel of experts and that any changes to the component interfaces that result from design decisions are agreed on by the Configuration Control Board.

__Implementation__ — The bulk of the work in this phase is the actual construction of the software modules and the creation of the release documentation (users' manuals, on-line help, operator's manuals, maintenance manual, etc.) and the test specifications. Unanticipated problems at this phase may cause changes in the design specifications and even the detailed interface requirements. If the interface specifications need to be changed these changes must be reviewed by the Configuration Control Board. Additional Design Reviews are held during this phase to review changes to the design and to have implementation walk-throughs of developed software. The QA responsibility in this phase is to ensure that adequate reviews are held, that CCB activities happen quickly and that a strong configuration control system supports the development activity and tracks each software module that is developed.

__Verification__ — This phase is the conclusion of the Configuration Item development cycle and all associated data are frozen and "delivered" with the software. This includes an audited version of the design specifications and a preliminary version of the Release Documentation (which now includes all the software configuration data). The activities in this phase involve the unit testing of all the Configuration Item's components and the creation of a configuration baseline that will be used during the integration test phase. As modules are unit tested, they are compared to the design specifications and any discrepancies are analyzed. The result of this design audit may be a change to the software or to the design information depending on the reason for the discrepancy. Enhancement Requests are generated if modifications to the system are desired.

__System Integration__ — This phase assembles the separate configuration item baselines and tests the integrated system for "end-to-end" correctness. In addition, the final work on the system validation and operational testing plans and specifications are completed. Any problems identified in the integration testing are recorded, categorized as Maintenance Orders or Enhancement Requests, and tracked to resolution.

__System Validation and User Evaluation__ — It is now time to get the real users of the system involved in the new increment of system functionality. A series of validation and evaluation tests are set up to test the system and its associated user documentation in a near operational environment with different users. The evaluation process can be supported by usability testing to judge the "ease of use" and other subjective criteria. This is done by measuring the efficiency of different users trying to execute specific system functions and analyzing performance bottlenecks.

This phase requires the baselining of the entire system, including the software, the release documentation, and all the supporting design and test data. All these together make up the Product Baseline which would be released to become an Operational Baseline if the results of the validation and evaluation were successful. However, some early increments are created just to get the feedback and performance evaluation from this phase and are not intended to become an operational system and in that case the development team would go back and create another increment after reevaluating the high-level requirements.

__Operational Testing__ — Once an increment is released for operational testing and placed in its final operational environment, any changes to the software and design data are carefully controlled. Each set of changes could require a retest of the entire system. The goal of operational testing is to assess the reliability of the developed software and the accuracy of the operations and users manuals, not the appropriateness of the applications (this should have been determined during validation and evaluation). Any problems during the test are recorded and tracked to resolution.

__System Operation and Evaluation__ — Once a release has passed operational testing, it is ready for actual operation. The important aspect of this phase to the TASC-EDGE life cycle is the opportunity to learn from the successes and failures of the system. Much attention should be paid to the experience and suggestions of the users in order to incorporate recommended modifications into future releases of the system.

SUMMARY

We have reviewed the quality assurance assumptions, goals, and results of the two most popular life cycle models: stepwise refinement (the "waterfall model") and incremental release (evolutionary development). The goals are quite different.

The waterfall model assumes that users are able to describe system requirements completely and consistently, hence the conclusion that as-specified requirements at the completion of requirements analysis are *nearly identical* to the users' real requirements. Throughout development, verification and validation activities are *biased against restating requirements*. When the original assumption about requirements is shown to be false, the waterfall model delivers a system that does not meet the users' real needs.

Evolutionary development assumes that users are unable to communicate requirements well, hence the as-specified requirements are *incomplete and lacking in detail*. Throughout the development process, quality assurance in the evolutionary model seeks to find opportunities for requirements refinement and enhancement. The opportunities are gained through evaluation and requirements respecification. There is a *strong bias towards restating requirements*.

The TASC-EDGE model addresses the requirements uncertainty problem. The model is a combination of classical life cycle activities and iterative enhancement at the development level. The use of development time-boxes, rapid prototypes, and user feedback allow TASC-EDGE to deliver systems that meet users' real needs. Complexity is layered into system by satisfying high-priority requirements in early releases, while continually managing schedule, budget, and system functionality.

Evolutionary development, when compared with classic waterfall development, has three quality assurance issues. The first is how to establish requirement baselines while actively soliciting requirement modifications. The TASC-EDGE model resolves this by defining two baselines: a Functional Baseline and an Increment Baseline. These baselines are reestablished when the development cycle iterates.

The second issue is applying configuration management techniques to separately evolving configuration items. In TASC-EDGE, a high-level steering group is responsible for the "direction" of system evolution. Lower-level CCBs manage more classical CM activities. Early freezing of interface specifications delimits boundaries between multiple CM levels.

The operational certification of a system built using an evolutionary life cycle is a third issue. TASC-EDGE employs a two-category discrepancy tracking method. This tracking, combined with corrective action, minimizes the risk of delivering defects. Also, multiple iterations of releases help to raise certification confidence by continually evolving validation test sets and ensuring testing thoroughness.

The TASC-EDGE model uses 11 development phases and various quality assurance activities and products. Several iterative loops are built into the development cycle, maximizing user feedback on incremental releases, to ensure that the delivered system meets users' real requirements.

REFERENCES

1. Boehm, B.W., "A Spiral Model of Software Development and Enhancement," Proceedings IEEE Second Software Process Workshop, *ACM Software Engineering Notes*, August 1986, pp. 22–42.

2. Brooks, F.P., "No Silver Bullet — Essence and Accidents of Software Engineering," *Computer*, April 1987, pp. 10–19.

3. Clark, P.G., Lobsitz, R.M., and Shields, J.D., "Documenting the Evolution of an Information System," *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference*, NAECON 1989, May 1989, Volume 4, pp. 1819–1826.

4. Lobsitz, R.M., "Growing an Information System Using the TASC-EDGE Methodology," *Proceedings of the IEEE 1988 National Aerospace and Electronics Conference*, NAECON 1988, May 1988, Volume 4, pp. 1328–1333.

5. Martin, C.F., *User-Centered Requirements Analysis*, Prentice Hall, New Jersey, 1988.

6. Martin, J., *Information Engineering*, Savant, London, England, 1986.

7. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, December 1972.

8. U.S. Department of Defense, "Military Standard, Defense System Software Development," DoD-STD-2167A, 29 February 1988.

9. U.S. Department of Defense, "Military Standard, Defense System Software Quality Program," DoD-STD-2168, 29 April 1988.

10. U.S. Department of Defense, "Military Standard, DoD Automated Information Systems (AIS) Documentation Standards," DoD-STD-7935A, 31 October 1988.

11. U.S. Department of Defense, "Military Standard, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs," MIL-STD- 483A, 4 June 1985.

Richard M. Lobsitz is the manager of the Information Systems Technology Department within the Software and Systems Integration Division at TASC. He has over 16 years of experience in building information systems in micro, mini, and mainframe environments. He oversees the application of the TASC-EDGE methodology to the development of information systems, decision support systems, and office automation systems for several of TASC's government clients. He received his BS degree from Washington and Lee University and an MS degree in computer information systems from Boston University.

Peter G. Clark has over 10 years of experience in software development and software engineering research. His current research at TASC focuses on software development methods and environments with a strong interest in tool integration issues. Mr. Clark was the principal investigator on the technical support contract for the Evaluation and Validation (E&V) of Ada Programming Support Environments (APSEs) and one of the major contributors to the E&V Reference System. He received his BA degree in chemistry from Cornell University and his MS degree in computer science from the State University of New York at Binghamton. Mr. Clark is a member of the ACM and SIGAda.

C. Robert String is a manager in TASC's Information Systems Technology Department. He has led various projects in information systems development, networked systems, and automated software testing tools. His research interests include software quality assurance, configuration management, and software testing techniques. He received his BS degree in computer science from Rochester Institute of Technology and is a member of the IEEE Computer Society.

SOFTWARE ENGINEERING EDUCATION "AN EXPERIMENT"

by
Pamela B. Lawhead
The University of Mississippi
and
Richard Hess
IBM Corporation

ABSTRACT

This paper describes a software engineering educational experiment undertaken at the University of Mississippi in conjunction with IBM Federal Systems Division in Boulder, Colorado. The experiment consisted of an effort between the two orgainzations to create and institute a course which reflected as closely as possible the industrial software development process. The course provided the students with the experience of creating a piece of software from specifications to code and final testing using a design methodolgy, a set of specification requirements and an Ada-like design language and most importantly a very tight development schedule which was non-negotiable. IBM provided the specifications, required the use of DOD-STD-2167A and their own Ada PDL for use in design. They also provided a liason with the University who met with the class for Requirements, Preliminary Design, Detailed Design and Code reviews. The undergraduate Junior Class was divided into five groups and the requirements were parceled out to each group who then met and produced the required documents for each stage in the lifecycle development process. One group was designated the integrating group and was dealt the responsibility of producing from the different pieces the final product to be given to IBM at each stage in the development. The process of the course provides many lessons for both academia and industry and lends itself to modeling by other institutions.

OVERVIEW

At the University of Mississippi in Oxford, MS we have developed a course in Software Engineering which uses Ada as its primary language and which requires the development of a large piece of software in a group environment. In past semesters the students have written Ada tutorials and a Motorola 68000 simulator. The students have done the development always on an IBM mainframe, sometimes using a 286 personal computer as a development environment because it provided a CASE tool on which to do the design. The decision to use a CASE tool was usually a function of the design methodology used. When the methodology was supported by the tool then the CASE tool was used. Recently a new twist was provided to the course which, while experimental, has proven to be beneficial to all involved.

HISTORY OF THE ADA COURSE

The Computer Science Department has been involved with Ada since late 1984 when it acquired one of the first compilers for an IBM mainframe. In 1985 we were involved in hosting and testing of the ALSYS IBM 370 compiler and have had use of many versions of that compiler since then. We currently have two different 370 Ada compilers, 286 compilers and a Sun compiler running along with EPOS CASE development tools. All of our compilers from ALSYS are running with their full toolset so that our students have very up to date software available. We also have Ada math libraries available on the mainframe. In this rich Ada environment our students, both graduate and undergraduate, are able to work 23 out of 24 hours per day on program development. This freedom has allowed them to become relatively sophisticated Ada programmers. They have now presented many papers on the results of their work in this environment, sharing with others the results of their insights.

In 1985 in response to the recommendations of our graduates and that of industry, we instituted a required undergraduate course in Software Engineering using Ada. Our purpose was to force our students to go through the lifecycle development process, in groups, while under deadlines and providing full documentation at every step of the process. The first four times through the course we required an Ada tutorial written in Ada, reasoning that in writing the text for the tutorial the students would "learn while doing". We have attempted, over the years, to teach Ada many, many different ways. We have taught it in traditional lectures as though it were a new language different from any other. We have had the students teach themselves with additional information provided by the instructor, we have used CAI tutors and we have just said "here is a book go and learn it".

We have now concluded that the most effective instructional method for our students is in traditional lectures but starting with the Pascal-like subset of Ada and moving quickly to the differences. The part of the language which the students have the most trouble with is the compilation dependencies. They have not had any previous experience with separate compilation so it is a new and difficult concept for them. Only one of the large projects developed thus far has required tasking so, even though we teach it we are in no position to evaluate our methodologies there. We have found though that starting with a previously written Pascal program, changing the syntax and compiling it on an Ada compiler is the fastest way to get them up and programming.

NEW FOCUS

This year we have tried a totally new approach to the course thanks to IBM Federal Sector Division. In the Fall of 1989 we entered into an arrangement which required that we produce for them a Statistical/Plot package in Ada during the course. This required that the class not only learn Ada but it also had to learn and use DOD-STD-2167, DOD-STD-2167A and IBM's FSD ADA STYLE GUIDE and had to follow the lifecycle development process presented to them in an IBM FSD inhouse document which outlined the exact format of each document required during the lifecycle. The actual design was done using IBM FSD's "Ada-Based Process Design Language (PDL/Ada-2)."

This was a risky venture on both sides, ours and theirs. We had to agree to complete a piece of software which was projected to be 3000 Ada lines (measured as ";" counts in non-commented code). The actual final hardcopy document presented was in excess of 7,000 lines of code, comments and headers and did not include the user's guide. We had no idea if it was possible to complete this work in the 14 weeks of a semester. We were working with a new group of students and some years they are better than others so we could not predict their ability to learn and use Ada. We also were not familiar with IBM's way of doing things so we did not fully understand the overhead which that would entail. We were a thousand miles away and that was potentially a problem. We did fully understand the IBM Mainframe Ada compilers and runtime environments so we knew exactly what we could and could not do in that environment. So our risk, while totally non-monetary was real and we were aware of it.
The risks incurred by IBM can be boiled down to the following four possible results:

1. After examining the System Requirements, The University of Mississippi (Ole Miss) would decide that the magnitude of the Plot Package exceed the semester time constraints.
2. For whatever reasons Ole Miss would never really get started on the project.
3. Ole Miss would deliver a product that only met a portion of the System Requirements.
4. Ole Miss would complete the project but without adhering to the coding standards set by IBM.

If a failure had been inescapable then option one or option four would have been most desirable. In the first case IBM would possess a considerable time cushion to implement a recovery plan. With option four IBM would have a working product which hopefully could be modified to meet programming standards. With the remaining two options, the risks to the delivery schedule would significantly increase. Both types of failures would have entailed a considerable effort in manpower to meet the critical review date, and the testing turnover date.

Happily none of the preceding events occurred and The University of Mississippi delivered the Plot Package on time, according to specification and with relatively few errors. It could be argued that with the commitment of the University's Computer Science Staff, coupled with the dedication and enthusiasm of the students there was very little risk that the project would not succeed.

The University agreed to provide IBM Federal Sector Division with a Statistical/Plot package which was to include:

1. A Correlation Function
2. Three Curve Fit Functions (Curve fit, Time, Step)
3. A Histogram Function
4. A Plot Function
5. A Statistical Sampling Function
6. A Set of functions to provide statistical summaries
7. An External Sort Routine
8. A Tabular Function.

The Package was to be written in Ada and run on an IBM 370. It was to be delivered on a tape and the code was to run in an MVS environment. We were going to have to write it in a VM/CMS (IBM's Conversational Monitor System - a proprietary operating system). We did not have access to MVS because it was running on a physically separate machine. While the two operating systems are very different we were sure that the conversion would be trivial and were confident of our ability to provide the Ada code and the interface to a different operating system. The code was to run on the IBM's own Ada Compiler which we had. We actually chose to develop it on the ALSYS compiler because the toolset was more comprehensive and we found the diagnostic error messages more meaningful. The output for the Plot package was to be displayed on an IBM line printer (model 3211-1) or on an IBM 3278 display device. We were given a delivery date and a review date for each required document.

IBM was to provide each student with a copy of the Requirements Specification and all related supporting documents. They were also to visit the campus once to review the requirements with the students,once for a preliminary design review, once for a detailed design review, once for a code review and finally for full acceptance of the product. It was hoped that we would be able to make some kind of networked connection with them so that we could download the intermediate documents to them and get quick feed back. As it waswe used overnight mail and FAX machines to do the communication. We set up a schedule which required us to give them each document approximately one week before they came to campus to visit so that we would maximize the effectiveness of each of their visits.

CLASS STRUCTURE

The class met two times a week for one hour and fifteen minutes each. The semester was 14 weeks long. Because we knew that we had real deadlines to meet we ran a graduate version of the class in parallel with the undergraduate class. We did this so that we could have some students who already knew Ada. Admission to the graduate class was by consent of the instructor. This assured us that we would have students who already had some Ada knowledge. The final enrollment included 21 students, five of whom were graduate students. The class was divided into five groups each headed by a graduate student. For the first two weeks of the class the emphasis was on Ada. At the end of six lectures the students had a brief test on Ada syntax and runtime environment issues. During this same time period the reading assignments also included the documents provided by IBM, especially two articles which they provided on IBM design methodology. These articles included "Using a Multi-level Design Method under DOD-STD-2167A" by Dr. Nancy R. Mead and Roger J. Lockhart and "Software Engineering and Ada in Design" by Don O'Neill. These articles gave the students a quick sense of the way FSD approached large Ada-based design projects. It also gave the instructor a springboard for presentation of the critical issues in the design of very large software projects.

Peter Lee and Rick Hess of IBM FSD arrived on Campus during the first week of February for a formal presentation of the requirements to the class. The students by that date were already in groups and were working on writing little programs in Ada in groups (A stack program, a program using enumeration types, and a sort program). They had been given group accounts and were in the process of setting up their virtual machines so that they had an electronic method of communication within each group. An office was set aside in the department for group meetings and each group was given a key to it. All critical documents were kept in this office. They had been briefly (the first ten minutes of each class) presented with interpersonal skill development exercises designed to enhance each group's ability to work together. These exercises included but were not limited to such "sharing" questions as "When I am angry the most productive thing for you to do is", "I work most effectively 1. alone, 2. with one person, 3. in groups...because" or "Three things which I do to stall getting down to serious work are...."

The class meeting before the IBM people arrived was devoted to our own Requirements Review. Each group was required to prepare and submit to the instructor questions about the Requirements Specification. These questions were reviewed, compiled and FAXed to IBM so that the time spent by FSD on campus would be maximally effective. The most critical problem which emerged was a problem with an MVS data set design for a very strongly typed language. This problem haunted us throughout the first half of the project, coming to be known as the "I/O problem" and finally taking on a life of its own. The Requirement Specifications specifically gave examples in terms of MVS data sets. Most of the questions prepared by the students focused on clarification issues. There were some actual duplications in the Requirements and the students located these. It needs to be stressed here that the Requirements were not perfectly specified at this point and that became a teaching tool which could not be duplicated in an "assignment" environment. In previous semesters of this class the requirements were created by the teacher and any lack of specificity could be handily corrected upon discovery by the students. This created an environment where the requirements could be negotiated and were, therefore, not taken seriously by the students. In this new environment the students worked very hard understanding the requirements and locating any ambiguities or lack of specificity because they understood that they were, for the most part, not negotiable. It cannot be stressed enough that the students learned about the importance of well-written requirements in a way that ten thousand lectures would never teach them.

For the Requirements Review IBM Representatives met with the class for an entire class period and then met with individual students after class and during the morning of the following day. By the time that they departed all problems except the I/O problem had been ironed out and the students could begin the preliminary design. The I/O problem remained a problem because the data to be processed was coming from yet another sub-contractor and, while the format was relatively fixed, the focus had to be changed from an MVS data set to an Ada file. It was agreed that the preliminary design would contain an I/O package which would contain our best efforts but that we would be allowed to change it after the preliminary design review. It became know to us as "Mystical I/O." The preliminary design was to be written in PDL/Ada-2 using a set of type specifications provided by FSD to enhance and insure portability. These were coded in an Ada package and "withed" into the design. Our preliminary design was submitted to FSD on the date specified (five weeks from the first class meeting). FSD was to compile it on their system, review it an then submit their review to us. Our students were to have compiled the preliminary design prior to its submission. Time got short and they simply ran the code through the parser and submitted it. When FSD attempted to compile it they got 29 errors all having to do with specification-body interface problems. When the students got that back they were horrified. After a very careful review of the process they corrected all of the errors and resubmitted the design. This time it compiled at FSD without error and was then available to them for preliminary design. It was a very sobering experience for them and served as notice that this whole thing was being taken very seriously by IBM. They did not fail again. In the original plan a PDL review was scheduled. As things progressed this was determined to be unnecessary. The PDL review was then done via telephone and written redlines. The only changes required had to do with "Mystical I/O" and the adherence to the stylistic conventions of an IBM internal document which we had not been given. The document was given to us, the changes made and the PDL was accepted.

Early during the Detailed Design period the "Mystical I/O" was finally fully clarified and a major design change by IBM was introduced. The system was to now be designed to run on a different machine (yet to be determined) but necessarily running Unix. Because our code was so totally modular and all I/O was completely buffered from every procedure this was a very minor change in our package. Much to everyone's surprise it required one change in the declaration of the I/O Body. The change had to do with the file naming convention. CMS uses a space between the name of a file and its extension, Unix uses a period. We had only to change the file separator from a space to a period in the declaration section of the body of the I/O package in order to make the code run on Unix. The detailed design was submitted on time, it was returned with redlines, the redlines after some discussion were corrected and the final detailed design was reviewed on campus by a representative from IBM.

The coding phase of the project, while hectic, was rather straight forward since the detailed design had been written in Ada. It took a while to get the error handling fully worked out and to get the I/O working so that all output was in a form which was accepted by all as very readable and according to specifications.

The project was completed on time with the final delivery and acceptance date changed from a Friday of one week until the Tuesday of the next. This was done by mutual consent of IBM and the University. On delivery date we received a Sun Compiler from ALSYS at about 11:30 a.m., we installed it in the afternoon, ported the code to the Sun from the IBM mainframe and the code compiled without error by 4:30 that evening. It did not run successfully because we were using a time stamp as a filename extension to keep our filenames unique. The difference in word sizes of the two machines prevented this from working because we could not achieve a time stamp granularity fine enough to work. We wrote a little algorithm which ensured the uniqueness of the time stamps and the code ran correctly. The students understood from this exercise exactly what portability was and became minor experts on the significance in software engineering of truly modular code.

BENEFITS TO INDUSTRY AND IBM

The most obvious benefit to IBM was receiving a completed, well documented Plot Package which was written to specification and delivered on time. There were also several non-quantifiable but no less tangible benefits to IBM. They were as follows:

1. The project placed IBM in a favorable light with future academic and industry computer professionals.
2. The project illustrated that IBM will utilize new, innovative methods and relationships in the production of software.
3. The project gave IBM employees who were supervising the project valuable experience and insight into the workings of the university computer science environment.

Finally, IBM provided the University with the opportunity to showcase its talents, and IBM increased the marketability of the computer science students by exposing them to an industry standard software development environment. These benefits are not only to the students and the reputation of the University, but to industry as well.

BENEFITS TO THE UNIVERSITY

The University benefited in many ways from this experience. Our students were exposed to the conditions of the real world of software production. They were forced to create a software package under constraints over which they had no control. They came to understand the Software Engineering Lifecycle in a way that no amount of lecturing, reading or programming in an ordinary academic environment could teach them. They gained experience in using industry standards which ordinarily are not even available to them. They were able to work in an environment where "almost" was not enough. They gained valuable experience in learning to depend on each other and to work out their differences quickly because the success of the project depended on their ability to work together effectively. The project was too big for any one person to simply take over and do it. It was necessarily a group effort. IBM very graciously gave each of them a personal letter thanking them for working on the project. They were able to add these to their resumes so that they could document their participation in a "2167A" project.

LESSONS LEARNED

Industry is correct when they say that students are not prepared to work on software engineering projects when they leave academia. Courses such as this one do prepare them fully to work on large projects, under tight standards and in groups. In choosing such a project the instructor needs to be fully familiar with all of the requirements of the project (documentation standards as well as code requirements). This course would have been better for the students if the instructor had actually written code under these guidelines. In the future, perhaps, industries which are preparing to participate in such projects should allow faculty to attend the workshops which they provide internally so that the faculty will be fully aware of the ramifications of the methodologies. It is difficult to fully appreciate how something is to be done from the outside. This experiment was successful at all levels and is to be recommended to all academic environments. It takes an enormous amount of energy to oversee it and it takes a level of work from undergraduate students which exceeds the normal requirements of a single semester course. However, the benefits to the students were so significant that not a single student complained about the work involved.

Pamela B. Lawhead is currently on the faculty of the University of Mississippi where she teaches courses in Programming Language Design and Software Engineering. She has been involved in teaching Ada courses since 1984. For further information she may be reached at The University of Mississippi
Dept. of Computer and Information Science
Weir 324
University, MS 39677
601-232-7396

Richard T. Hess, Jr is an applications programer with IBM Federal Sector Division. He is currently responsible for a textual user interface written Ada on the RISC 6000. For further information he may be reached at:
IBM MC(002C)
Dept. TN2
6300 Diagonal Hwy.
Boulder, CO 80301-9023
303-924-4132

Leveraging CBT in Universities to Produce Productive Ada Students for Industry

James E. Walker

Network Solutions Incorporated
505 Huntmar Park Drive
Herndon, Virginia 22070

Abstract

Computer-Based Training (CBT) is not "bleeding edge" technology, nor is it by any means a panacea to all of todays training problems. However, when used as an integral part of technology education and training process, computer-based training can be instrumental in providing individualized instruction; imparting and verifying the transfer of cognitive data; providing remedial and reinforcement instruction; providing drill and practice through exams and programming assignments; and by supplementing classroom instruction to enable instructors to make optimal use of classroom time.

This paper will indicate industry's expectations from Ada software engineers; identify a traditional university process for educating and training students; and illustrate various processes for integrating Ada CBT into traditional university classroom environments. The proposed processes should help improve the readiness of students entering the Ada software engineering industry.

Industry's Expectations from Ada Software Engineers

The ideal candidate for a typical Ada software development project would hold the following credentials:

- Working knowledge of Ada's syntax, semantics, and constructs
- Working knowledge [illegible] Ada-oriented design and development methodologies (e.g., OOD)
- Working knowledge of software lifecycle activities
- Ada project development experience where Ada's software engineering strengths were exploited
- Working knowledge of NASA and DoD software development and quality assurance standards (e.g., DoD STD-2167A and DoD STD-2168)
- Experience in software process engineering
- Experience in building reusable software components
- Experience in using CASE tools

Given that the total average classroom time allotted for a 3 credit hour university level course is about 52 hours, it becomes apparent that industry's demands for Ada competent graduates are a bit overwhelming. Current undergraduate programs in Computer Science are insufficient to meet the demand for trained personnel in the software engineering industry [Warner, 1989]. As a result of the limited amount of classroom time allotted for instruction, most courses focus on Ada's features and not on Ada project development. It is imperative that students become equally *trained* in Ada as they are *educated*. A reliable axiom indicates that we recall 25% of what we "hear", 45% of what we "hear and see", and 70% of what we "do".

Traditional Ada Education and Training Process

Fifteen years ago Computer Science was virtually a new discipline in university curricula. Programming language courses of that era were constructed with heavy emphasis on the instruction of syntax, semantics, structures, algorithms, and basic programming concepts. Most freshman computer science majors during that time had not been exposed to computers, let alone programming languages. University instructors had to perform substantial amounts of hand-holding while introducing students to this new discipline.

A Greater Awareness in Structured Programming

Given the advent of affordable computer technology, many freshman computer science majors have opted to purchase their own personal computers. Studies show that over 50% of freshman computer science majors have acquired programming experience in at least two structured programming languages (generally BASIC and Pascal) partly to satisfy their technical curiosity and partly because many high schools are currently teaching structured programming. Although students now have a better foundation for university instructors to teach more software design and development concepts, many syllabi for programming language courses identify inordinate amounts of c!assroom time for laying again the foundation of programming fundamentals.

Traditional Process for Educating and Training Ada Students

In many universities the teaching of Ada has been patterned after traditional university programming course infrastructures. The following is an outline of a traditional process:

- Teach Ada's syntax, semantics, and constructs in the classroom
- Issue small, out-of-class programming assignments to be completed in the lab
- Give written exams that qualify students' knowledge of Ada's features

In exceptional cases, instructors will attempt to enforce good software engineering practices on the small assignments. This is often difficult when the student has not been introduced to some of Ada's more salient software engineering features, such as *Packages* and *Generics*. Many instructors choose to postpone these topics until later on in the course or in an advanced course. This traditional approach to educating and training our students caters to programming in the small, but not in the large.

Another dilemma with our traditional educational process is the spurious mindset that's being created in the students. Throughout the traditional process, students are generally rewarded according to their individual efforts. The software development industry is not as concerned with individual effort as they are with team effort. For example, if a software project manager has a 10 member software development team and the test engineers fail to perform their role thoroughly, then how understanding can we expect the customer to be if the manager deploys an erroneous or non-compliant software system? Although the customer may recommend that the test engineers be replaced, the customer will ultimately hold the company responsible for deficiencies in the software.

Revised Ada Education and Training Process #1 (Leveraging CBT for Software Project Development)

Since students today now have a greater programming awareness, we must update our university programming language syllabi to take advantage of the students new knowledge. In particular, we can leverage CBT to enable instructors to better utilize classroom time by focusing on the software engineering aspects of Ada, versus the instruction of programming fundamentals. Software engineering is concerned with the methods, tools, and techniques used to develop and maintain computer software. An appreciation for, and understanding of, software engineering concepts is best gained by applying them to a real software project [Fairley 1985].

The following is a revised Ada educational and training process that leverages CBT for Software Project Development:

- Let the CBT teach Ada's syntax, semantics, constructs, Ada-specific features, and other cognitive details
- Let the CBT issuc small programming assignments at an individualized pace
- Let the CBT give objective exams and verify students' knowledge of Ada's features

- Let the CBT generate progress reports to assist the instructor in identifying requirements for remedial assistance
- Let the instructor conduct an overview of software engineering with Ada during the first half of the course (e.g., 1/2 semester)
- Let the instructor conduct an Ada workshop during the second half of the course
- Let the students keep the CBT as a reinforcement tool for after course/pre-employment practice, as they would any textbook

Students should also acquire Computer Aided Software Engineering (CASE) tool skills. CASE tools have proven to be useful by automating various portions of the software lifecycle. As a note of information, there is a shareware CASE tool for drawing Data Flow Diagrams, Transformation Schema, State Transition Diagrams, Structure Charts, and Entity-Relationship Diagrams in accordance with the Yourdon-DeMarco, Gane & Sarson, Ward-Mellor, Hatley-Pirbhai, Yourdon-Constantine and Chen methods for structured systems analysis and design (including real-time systems). This tool can be obtained from Evergreen CASE Tools, (206) 881-5149.

To supplement/complement the CBT portion of the course, the following is a proposed timeline for the revised process:

I. Overview of Ada (26 Hours)

A. The Software Crisis & History of Ada (2 Hours)

B. Object-Oriented Design (2 Hours)

C. Ada from the Top-Down (3 Hours)

D. Data Abstraction and Ada's Types (2 Hours)

E. Subprograms (2 Hours)

F. Packages (3 Hours)

G. Generics (3 Hours)

H. Tasks (3 Hours)

I. Exception Handling (2 Hours)

J. Input/Output (2 Hours)

K. Low-Level Features (2 Hours)

II. Ada Workshop (26 Hours)

A. Overview of the S/W Proj. Deve. Infrastructure (3 Hours)

B. Organize Team(s) (2 Hours)

C. Assign Projects, Issue Documentation Formats (2 Hours)

D. Develop Preliminary Requirements Document (2 Hours)

E. Develop Preliminary User's Manual (2 Hours)

F. Deve. Exter. & Arch. Design Specs for Prototype (2 Hours)

G. Conduct Preliminary Design Review (2 Hours)

H. Begin Det. Design & Implem. for the Prototype (3 Hours)

I. Develop Test Plan for the Prototype (2 Hours)

J. Test Prototype (2 Hours)

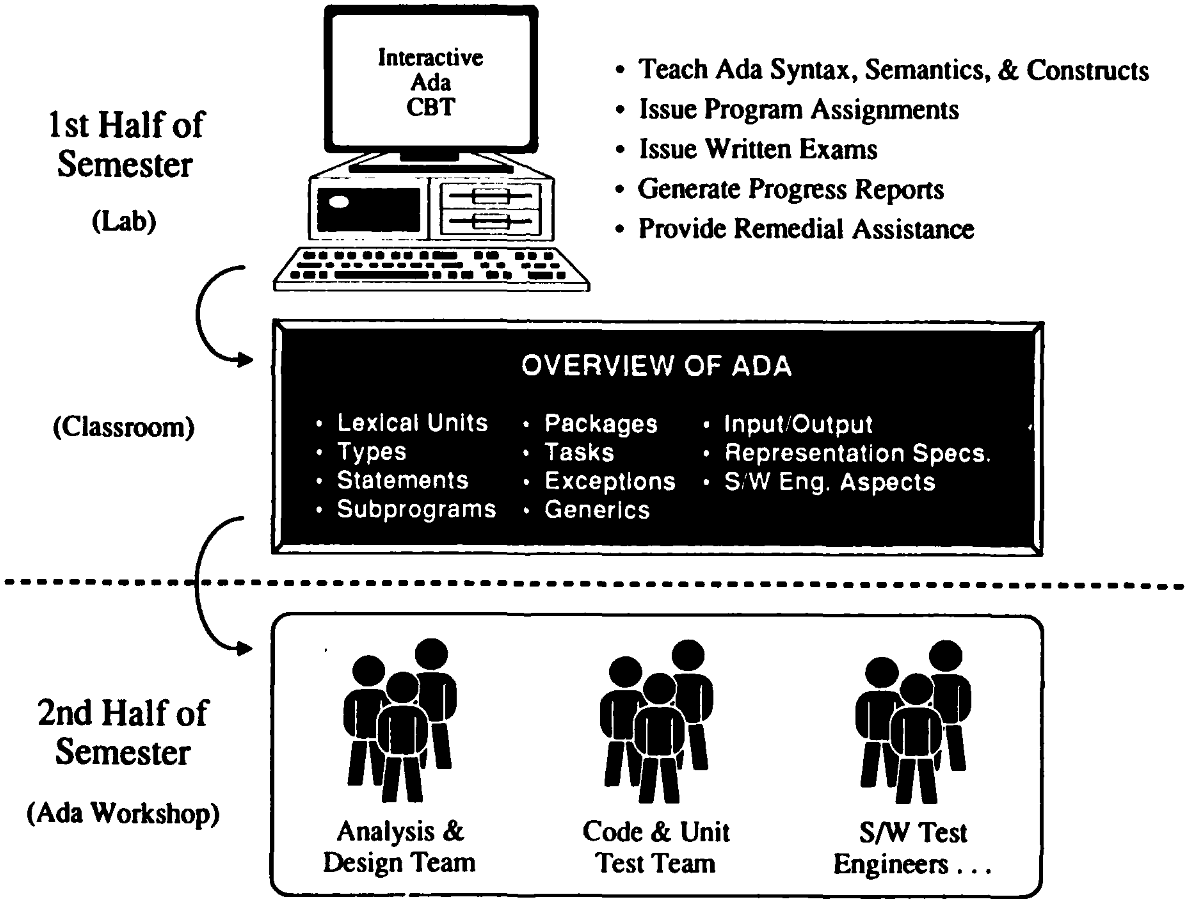
K. Demonstrate Prototype (4 Hours)

The above workshop outline is a subset derived from Richard Fairley's recommendation for term projects in his "Software Engineering Concepts" book. Dr. Fairley's book is an excellent guideline for instructors who are considering implementing a software design and development course.

Revised Ada Education and Training Process #2 (Leveraging CBT for Software Maintenance)

One of the first assignments generally given a new employee in the software industry is to maintain an existing large software system. The paradox of this assignment becomes obvious given that college graduates don't have a wealth of experience in programming in the large, let alone maintaining another programmer's software. This is probably one of the most challenging feats for any software engineer. One must first gain understanding of the software requirements and then wade through a previous programmer's logic before making modifications to provide enhancements to the software; adapting the software to new environments; or correcting problems with the software.

Leveraging CBT in the Classroom



1st Half of Semester

(Lab)

Interactive Ada CBT

- Teach Ada Syntax, Semantics, & Constructs
- Issue Program Assignments
- Issue Written Exams
- Generate Progress Reports
- Provide Remedial Assistance

(Classroom)

OVERVIEW OF ADA

- Lexical Units
- Types
- Statements
- Subprograms
- Packages
- Tasks
- Exceptions
- Generics
- Input/Output
- Representation Specs.
- S/W Eng. Aspects

2nd Half of Semester

(Ada Workshop)

Analysis & Design Team

Code & Unit Test Team

S/W Test Engineers . . .

The Software Engineering Institute provides a document entitled: "Software Maintenance Exercises for a Software Engineering Project Course" that comes with Ada source code for a Documented Ada Style Checker (DASC) software system. The document provides maintenance exercises that will enable students to gain working knowledge of maintenance aspects of the software lifecycle. Students will gain experience in developing documentation standards, configuration management plans, and regression test plans. Students will also work with others to enhance and correct the DASC software system.

Conclusions

By offloading the language-specific features of Ada to an interactive CBT, instructors can now spend quality time in the classroom teaching software system development. Instructors can issue statements of work to the students and let them analyze requirements and develop a requirements document. By emulating a contract environment, students can gain experience in participating in design reviews and acceptance testing. Students will soon realize that the customer doesn't always understand what he/she wants. Students can experience the frustration of engineering changes and how they impact cost and rework.

We do not propose CBT as a substitute for human interaction, but rather as supplemental aid for faculty and students. Ada CBT is also an ideal tool for students taking courses that heavily use Ada for illustrations, but do not formally teach the language. Courses that fall in this category are typically Data Structures, Software Engineering, and Operating Systems.

I should mention that in my opinion, industry has a significant responsibility to work with faculty to provide them with the necessary software project exposure and experience prior to expecting universities to produce "Productive Ada Students".

References

Engle C., Ford G., Korson T. Software Maintenance Exercises for a Software Engineering Project Course, Software Engineering Institute, 1989.

Fairley, R. Software Engineering Concepts, McGraw Hill, New York, NY, 1985.

Warner K. Integrating Ada into the University Curriculum: Academia and Industry - Joint Responsibility, Fourth Annual ASEET, 1989.

AN ADA-BASED TRANSLATOR WRITER SYSTEM LANGUAGE

Thomas F. Reid

Contel Technology Center
Chantilly, VA

Abstract: Traditional translator writer systems use Backus Normal Form (BNF) to specify the syntax of a language, and attributes and semantic actions to specify the semantics. The Ada Translator Specification Language (ATSL) extends this to a readable structured special-purpose language. ATSL uses extended BNF (EBNF) for syntax with embedded Ada code fragments for semantic actions; inherited, synthesized, and local attributes for internal communication; and with and use clauses for external communication. The Ada Translator Writer System (ATWS) recognizes ATSL and produces a stand-alone recursive descent parser/embedded semantics package in Ada and a parser/scanner interface package identifying the tokens, keywords and special symbols. This is combined with skeletal main, scanner, and symbol table packages.

Introduction

The motivation for this paper is helping the implementor quickly develop and understand the specification of a target language to a translator writer system (TWS). Easy to develop in that a rich number of abstractions are handled. Easy to understand in that the format is close to a traditional programming language. This work is seen as the first step in the specification of a special purpose translator language.

Traditionally, TWSs have been used to build production compilers. However, there is a growing awareness that the technology can have productivity impacts on building the "little" text-to-text translators in one's daily environment - the kinds where one has to convert the format of a document or file.

Potential TWS applications include constructing tool-to-tool interfaces for software development and end user environments; generating documentation, standards checkers, analyzers, and metrics from source code and designs; and building aids such as screen painters, test scripts, and automatic generation of makefiles.

TWSs can be used anywhere there is repetitive data entry and the tool saves more than the overhead of building the tool. The key is discovering situations where repetitive tasks are frequent. The TWS can then abstract the commonality and provide the user with an input language to express the unique.

TWSs are an enabling technology for the final stage of the reuse spectrum which starts with the first time anyone has built a program for an application domain, proceeds through libraries of explicit abstract data types, to being implicitly embedded in the data types, operators, and statements of a language. The reuse libraries are not lost since the language provides the "glue" to instantiate and manipulate the ASTs.

Translators

The Ada Translator Writer System (ATWS) and the Ada Translator Specification Language (ATSL) in this paper assume a generic architecture for a translator (Figure 1). A complex translator will have all of the components. A simpler translator may omit or combine components.

The scanner (or lexical analyzer) accepts a character at a time and produces a word or token at a time while verifying the validity of the tokens. The parser (or syntax analyzer) accepts a token at a time and produces a parse or syntax tree and a symbol table while verifying the syntax of the language. The semantics analyzer uses the parse or syntax tree and the symbol table to determine whether a syntactically correct source program makes sense. Finally, the synthesizer produces the output language. Both the semantic and synthesis phases may make multiple passes over the parse/syntax tree.

Translator Writer Systems

A TWS speeds the production of a *target* translator. A TWS accepts the specification of a target translator (e.g.,

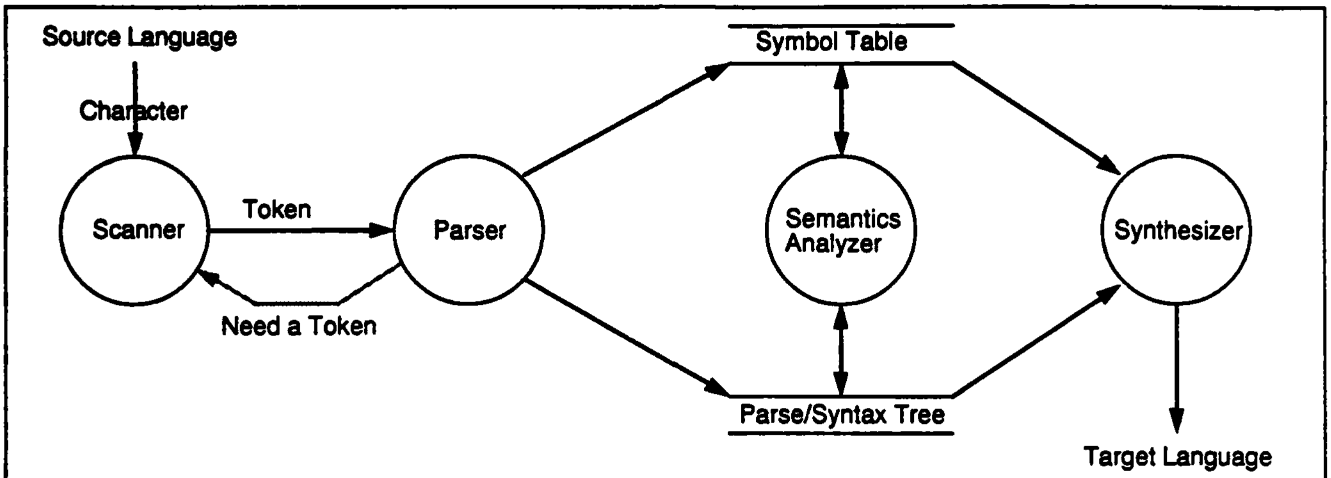


Figure 1 - Generic Translator Architecture

BNF and semantics for Pascal) and then produces a Pascal translator. The Pascal translator then translates Pascal source code to its target language (e.g., p-code or object code). The ultimate trial a TWS must pass is producing itself.

Thus, a TWS is itself a translator. Figure 2 shows the relationship between a TWS and the target translator it builds. The generic architecture of Figure 1 holds for a TWS as well as the target translator it builds.

The traditional TWS uses an attributed grammar for the front end to describe the syntax analysis, semantic analysis and synthesis. An attributed grammar intermixes the syntax of a language - in our case, extended Backus-Naur Form (EBNF) - with semantic actions, which direct the semantic analysis and synthesis. The synthesis step performs the target language (code) generation. In a one-pass translator, the parsing, semantic analysis and synthesis phases are performed in parallel.

For multi-pass translators, the parser builds an intermediate form (normally a parse or syntax tree) which is traversed by the back end to complete the semantic analysis, intermediate code generation, machine independent and dependent optimizations, and target language generation (one or more of the above phases according to the language definition and requirements). The back end is generally handcoded. MetaTool[1] and at least one other TWS (Q-Parser[4]) provide tools or a 4GL to help you walk ASTs.

Top-Down Versus Bottom-Up Parsers

Parsers can either be top-down or bottom-up. If you think of a parse producing a tree (like diagramming a sentence in high school English) where the leaves of the tree are the source program, the question is whether you produce the tree from the leaves up to the root or from the root down to the leaves. A top-down parser starts at the root and works downward to the leaves of the tree

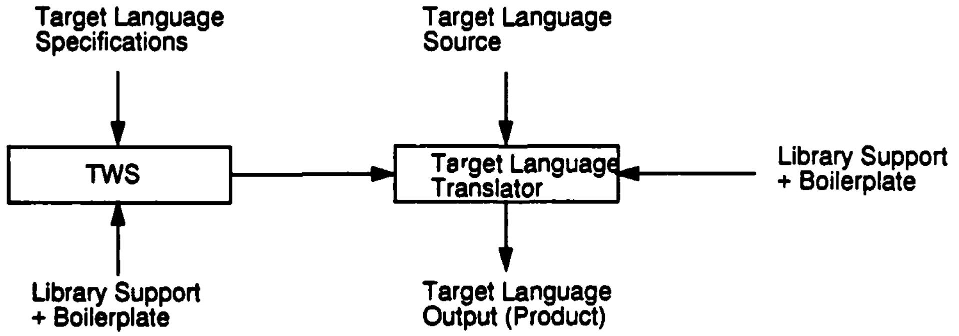

Target Language Specifications
Target Language Source
TWS
Target Language Translator
Library Support + Boilerplate
Library Support + Boilerplate
Target Language Output (Product)

Figure 2 - Relationship Between a TWS and Its Target Translator

while the bottom-up begins at the leaves of the tree and builds the tree working up toward the root.

Traditional production-quality TWSs are usually bottom-up because their LALR grammars are cleaner than top-down's LL grammars and error recovery is easier in table-driven bottom-up parsers. Traditional top-down parsers have had the advantage that inherited attributes are easier to implement.

Top-down recursive descent parsers have traditionally suffered because the LL1 grammars could not handle left recursion which is natural for left associative operators. Translating left recursion to right recursion made the grammar harder to understand and insert semantics. However, the discovery of how to produce a recursive descent parser directly from EBNF has eliminated this problem[2] and EBNF grammars for top-down parsers are more concise and intuitive than their LR bottom-up equivalents.

Error recovery is still a problem because most top-down parsers normally produce HOL source code for the translator rather than the bottom-up's usual tables. Information in a table-driven parser's stack is easier to manipulate than the implicit stack of a set of recursive procedures.

Proposed Ada Translator Specification Language

Our goal is to evolve a TWS specification language into a full special purpose language. To keep from reinventing the wheel, the cleanest way is to embed the facilities of a base programming language. A model is the Ada-SQL interface. Thus, the ATSL will can be loosely thought of a preprocessor for Ada.

In actuality, the amount of translator information overwhelms the Ada. It is the opposite of having occasional fragments of foreign code embedded in Ada. Ada influences the overall structure of the ATSL and Ada code fragments are used for the semantic actions. For an example, see the specification in Figure 3. Also, see Appendix A for a listing of the EBNF for the ATWS.

The ATSL has been designed with the following principles:

- Extended BNF (EBNF) with closure, option, alternation and grouping. These constructs allow the number of productions of the grammar to be reduced by at least an order of magnitude and makes the BNF more understandable. The resulting target recursive descent translator will have one procedure for each EBNF production. The ATWS takes care of calculating the embedded FIRST and FOLLOW sets (i.e., the selector set) of the implied LL1 grammar.

  Braces ("{" and "}") are used for closure which indicates zero or more repetitions of what is enclosed. Brackets ("[" and "]") indicate that what is enclosed is optional. The vertical bar ("|") is used for alternation which separates possible choices. Parentheses may be freely used. Figure 3 has examples. Appendix A shows the syntax of how the constructs interact.

- Embedded semantic actions. The semantic actions are Ada code fragments which are interspersed with the EBNF symbols. The remainder of any line with a "!!" (configurable through a directive) is treated as a semantic action. This is shown in Figure 3 where the EBNF is on the left hand side of the listing and the semantic actions are on the right. Since the scanner for ATWS is column insensitive, this is a style to make the listing more readable.

  In a single pass translator, the semantic actions determine whether the source language makes sense, build the symbol table (if needed), and perform the translation to the target language. When the parser is done, the translation is done.

  In a multi-pass translator, the semantic actions can do as much validation and synthesis as possible during the parse, but they must also set up the structures (such as the symbol table and parse/syntax trees) for the passes that follow.

  In the present ATWS, the semantic actions are captured by the scanner, sent to a temporary file, and then merged into the recursive descent parser during synthesis. The Ada code fragments are not validated in any way. This is left to the Ada compiler which builds the target translator.

- Declaration of interface packages. The ATWS builds a "stand-alone" parser/semantic action package that can be compiled and linked into the target translator without being edited. Good development practice is the semantic actions driving abstract data types such as those for the symbol table and the parse/syntax tree. The semantic actions should specify *what* should be done, not *how*. ASTs can be developed quickly for prototyping and tightened later for development.

  Figure 3 shows the declaration of the interface packages between the specification header and the declaration of the attributes.

- Declaration of inherited, synthesized and local attributes. The ATWS builds one procedure for each production (nonterminal) of the target language. Any time it encounters a nonterminal on the right-hand-side of a production, it calls the procedure for that production.

  Information (termed *attributes*) can be passed between productions as parameters of the procedure calls. An attribute is *synthesized* if it is passed from the called procedure back to the caller, and *inherited* if passed from the caller to the called. For an attribute, the ATWS declares a local variable in the calling procedure and passes the variable as a parameter in the procedure call. In the called procedure, the corresponding formal parameter is pass-by-value if it is inherited and pass-by-referenced if synthesized. The local attribute is an extension which allows a production procedure to declare a local variable.

Each attribute is declared as a four-tuple giving the name of the attribute, its (Ada) data type, the nonterminal to which it is attached, and whether it is inherited, synthesized or local. Lines 3-8 of Figure 3 are the attributes for a fragment of an integer calculator.

- Correspondence of terminal symbols with identifiers. Since the ATWS has been developed as a teaching tool, it is important that the parser listing be readable. The ATWS builds an enumerated scalar of the kinds of tokens in the target language as part of the parser/scanner interface. Without some help, the enumeration constants are artificial. ATWS has a lexeme declaration section which maps the values of the tokens to a name. Lines 9-13 of Figure 3 shows the syntax. Without the lexeme declarations, the semantic action fragments like 'Sign = SSPlus' would look like 'Sign = TK043' and the 043 would change as the language changes.

Translator Package Hierarchy

There is a generic package hierarchy for a translator which mirrors the generic translator architecture. They are broken into three types of packages (Figure 4):

- Skeleton packages. The Main, Scanner and GetChar packages vary little in all translators. Main produces the translator's greeting, prompts for and opens input and output files, calls the parser and other phases, and closes down the program.

  ATWS uses a generic scanner which recognizes keywords and special symbols whose definitions are supplied by the automatically generated interface package. The scanner also has "standard" code for identifiers and number systems as well as handling the printout and directives.

  GetChar provides a character at a time to the scanner. There are interactive terminal and batch file versions. GetChar also handles directives.

- Generated packages. ATWS generates the parser with the embedded semantic actions and the interface package integrating the parser and scanner.

- User developed packages. Each translator has processing which is unique and must be developed. These can be as separate passes or support to a simple-pass translator. If these packages are developed carefully they may be reused. Examples are symbol tables and parse/syntax trees.

```
translator Calculator;
with INTEGER_IO; use INTEGER_IO;
with Scanner; use Scanner;
attributes
   syn Expr . Val = INTEGER ;
   loc Expr . Sign = TokenKinds;
   syn Term . Val = INTEGER ;
   loc Term . Sign = TokenKinds;
   syn Factor . Val = INTEGER ;
lexemes
   SSPlus = "+" ;
   SSMinus = "-" ;
   SSMult = "*" ;
   SSDivide = "/" ;
begin
Lang  ::= Expr          !! Put (Expr_Val);
Expr  ::=               !! Sign := SSPlus;
           [ "+" | "-"  !! Sign := SSMinus;
           ]  Term      !! if Sign = SSPlus
                        !! then Val := Term_Val;
                        !! else Val := - Term_Val;
                        !! endif;
           { ( "+"      !! Sign := SSPlus;
             | "-"      !! Sign := SSMinus;
             ) Term     !! if Sign = SSPlus
                        !! then Val := Val + Term_Val;
                        !! else Val := Val - Term_Val;
                        !! endif;
           } .
Term  ::= Factor        !! Val := Factor_Val;
           { ( "*"      !! Sign := SSMult;
             | "/"      !! Sign := SSDivide;
             )  Factor  !! if Sign = SSMult
                        !! then Val := Val * Factor_Val;
                        !! else Val := Val / Factor_Val;
                        !! endif;
           } .
Factor ::= . . .
 . . .
end Calculator .
```

Figure 3 - Example ATSL Specification

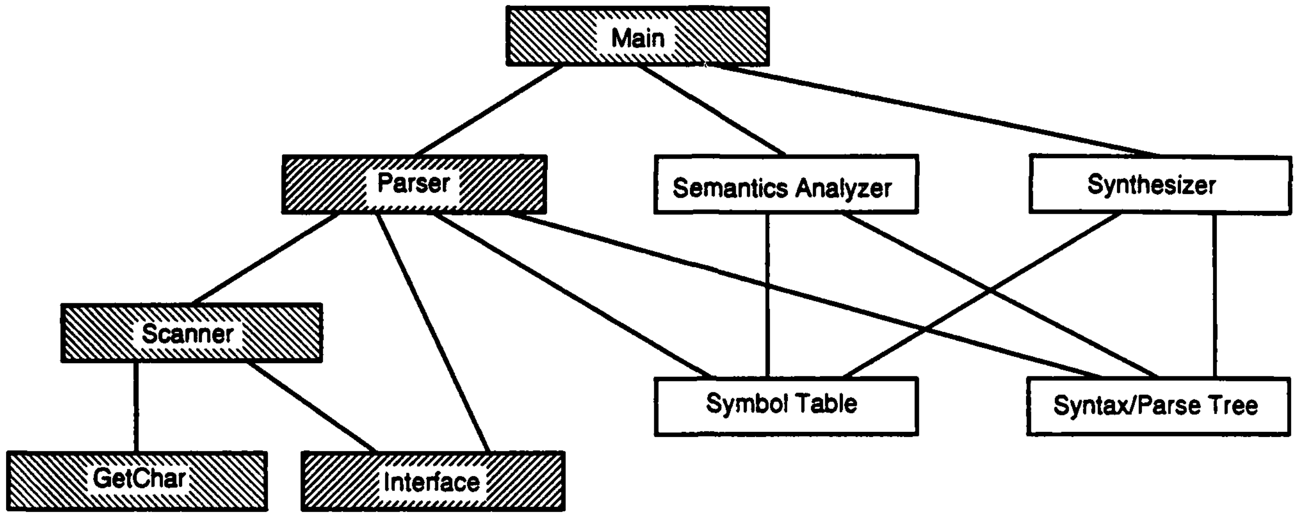


Figure 4 - Generic Module Hierarchy

ATWS Evolution

The concepts have been developed over 10 years of teaching translator construction courses in which the students generally modified a previous classes' translator writer system and then used it to implement small language compilers. The implementation languages have evolved through various Pascals to Modula-2 and is in the process of evolving to Ada. The translator writer systems have evolved from an LL1 parser generator through adding semantic actions, attributes, and with/use clauses to providing a stand-alone translator package.

The ATWS will be able to make itself. In fact, that is how it will be produced. It is presently implemented in Modula-2 and being used in a translator construction class. An experienced Ada student's project is using it to produce a Modula-2 to Ada translator which will be then be used to automatically rehost the TWS.

Long term plans are to develop a rich set of packages to support translator development such as a canonical parse tree abstract data type. The ATWS will then have an option to automatically build a canonical parse tree and the abstract data type will provide tree walking and reporting capability.

Appendix A
The TWS EBNF

```
EBNF ::= Header { Imports } { Attribute } { Consts }
         begin Prod { Prod } end Identifier .

Header ::= translator Identifier ";" .

Imports ::= with Identifier ";" [ use Identifier ";" ] .

Attribute ::= attributes { ( syn | inh | loc )
              NonTerminal "." Identifier "=" Identifier ";" } .

Consts ::= lexemes { Identifier "="
           ( NonTerminal | SpecSymbol | Identifier ) ";" } .

Prod ::= NonTerminal "::=" Expression "." .

Expression ::= Term { "|" Term } .

Term ::= [ SemAction { SemAction } ] { Symbol } .

Symbol ::= (  NonTerminal
            | SpecSymbol
            | Identifier
            | "(" Expression ")"
            | "[" Expression "]"
            | "{" Expression "}"
           )  [ SemAction { SemAction } ] .
```

References

1. AT&T, *MetaTool™ Specification-Driver-Tool Builder User Manual,* Release 1, June 1990.

2. William A. Barrett, Rodney M Bates, David A. Gustafson, and John D. Couch, *Compiler construction: Theory and Practice,* 2nd Edition, Science Research Associates, 1986.

3. J. G. P. Barnes, *Programming in Ada,* 2nd Edition, Addison-Wesley, 1984.

4. QCAD, QPARSER+ Translator Writer System Marketing Literature, QCAD Systems, Inc., San Jose, CA.

5. Niklaus Wirth, *Programming in Modula-2,* 3rd Corrected Edition, Springer Verlag, 1985.

Speaker

Dr. Thomas F. Reid is a Senior Menber of the Technical Staff at the Contel Technology Center in Chantilly, VA leading research and development into software methods, tools and environments. He is also Adjunct Professor of Computer Science at Virginia Tech. Dr. Reid has held positions at the Software Productivity Consortium, the MITRE Corporation, McDonnell-Douglas, and IBM. He has taught more than 15 industrial Pascal courses; graduate-level courses in compiler construction and formal languages; and Professional Development Seminars in Modula-2 and Pascal. Dr. Reid received his Ph.D. in computer science from the University of Southwestern Louisiana in 1978, his M.S. in mathematics from the University of North Carolina in 1964 and his B.A. in mathematics from Western Michigan University in 1962. Dr. Reid is a member of ACM, IEEE Computer Society, the IEEE/MSC/P1151 Modula-2 Standards Committee and ACM SIGAda, SIGSOFT and SIGPLAN. He has been Chair of the St. Louis Chapter of ACM and Chair of the Washington DC ACM Chapter Professional Development Committee.

QUALITY ASSURANCE USING Ada AND DOD-STD-2167A

David Disbrow *David Martin*

TELOS Systems Group
Lawton, Oklahoma

ABSTRACT

This paper discusses the Quality Assurance (QA) approach to software development indicating its methodology using Ada and DOD-STD-2167A.

INTRODUCTION

QA is a pre-established set of instructions or procedures which when applied, act as an "extra set of eyes" to monitor, review, evaluate, and audit the complete software development cycle to determine the quality of all associated processes and products from receipt of requirements to system final delivery. However, with the introduction of Ada and DOD-STD-2167A, older ideas of QA must be continually evaluated to conform to these new directives. QA specialists no longer just monitor walkthroughs and reviews, but become a part of them because of the new requirement to present a formal status of documentation and software to the customer during formal reviews. Internal problems are solved before presentation to the customer, causing the QA role to change significantly. This paper will detail this approach used by the Quality Assurance personnel.

PROJECT PERSPECTIVE

The Quality Assurance Element must be an independent activity reporting to a manager separate from the software development activity. Concerns must be surfaced and resolved as early in the software development cycle as possible. During the software development cycle, the earlier concerns are detected, analyzed, and implemented, time and effort loses are minimal. The longer problems go undetected, the more costly and time consuming they are to correct. It is our policy to handle all concerns at the lowest level possible; however, if necessaiy, any concern can easily be elevated to the PM level for necessary resolution. With this concept in mind, all interim concerns are normally resolved prior to formal reviews with our customer.

QA APPROACH

QA personnel use both a direct and indirect approach to accomplish their job. Normally the direct approach is preferred and is operationally friendly. The indirect approach is much less friendly but usually uncovers a majority of concerns. Explanation and examples of both approaches are necessary.

The Direct Approach: This QA technique involves face-to-face conversations and causes software developers to clarify reasons why design or documentation is accomplished in a certain way. This must be a two-way conversation and both parties must be flexible. It is extremely important that the QA specialist be fully prepared for the conversation. Associated documentation (DOD/MIL standards and company policy and procedures) must be readily available to support the QA position. The Quality Assurance Section is not an enforcing agency, but a source of information and an aid to management. The other side of the conversation is always extremely important. Normally, there are specific reasons an item must be completed outside specified guidelines and QA support is vital. Once both parties agree, a waiver or deviation is prepared (if necessary). QA will prepare this documentation and submit it to the PM for signature. This allows the software development group to proceed without the burden of administrative paperwork. Also when QA submits a deviation/waiver to the PM, this is a clear indication both parties agree and this is the best approach under the circumstances.

Indirect Approach: Other techniques used by QA are monitoring, reviewing, auditing, and reporting. These techniques, when implemented, use the indirect approach. These methods need to be discussed because they will be referred to later in this paper.

Monitoring. By observing, listening, and checking, QA verifies adherence to requirements of all applicable policies/procedures and standards. The monitoring of activities helps ensure that areas of concern are observed early in the software life cycle so that appropriate corrective action may be taken.

Reviewing. Reviewing is a method to verify compliance with company policies, governmental standards, software documentation, developmental libraries, programmer documentation, and source code. A periodic review is also made of company policy and procedures to keep them current with governmental policies and standards.

Auditing. Auditing is an independent activity which determines through investigation the adequacy of established procedures and compliance with them. Each audit is planned to correspond with the development schedule for the records being audited. Sampling techniques are sometimes used during the initial audit. If any problems are uncovered by the initial audit, a more thorough investigation will be undertaken. QA will conduct both formal and informal audits.

Reporting. QA policy and procedures contain checklists for required activities during each phase within the developmental cycle. A checklist will be completed and maintained for all products and the associated processes required to complete the product. Each time a QA technique is applied, a checklist will be completed and if concerns are noted, the affected person/group is given a copy. Follow-up checks are mandatory.

Using the direct approach, QA gives a more positive approach to their working relationship with other people. The indirect approach causes people to feel the QA specialists are not working as a part of the team. This, of course, is human tendency and is not the case.

TRANSITION

QA assumed some new project responsibilities during the transition to DOD-STD-2167A. The software task groups were busy studying all of the aspects of Ada. Portability, usability, and modularity were becoming common words throughout the work place. Some gaps began appearing and the Quality Assurance Element assisted by preparing a DOD-STD-2167A Quick Reference Guide. This assisted all aspects of the project by defining specific responsibilities. Software groups could easily determine what documentation would be required and when and how the document is to be delivered. This was also true for system testing, configuration management and QA. Summaries of all corresponding Data Item Descriptions and new terms were clearly explained. Over a period of one and one half years, we had not only trained our employees in Ada but also became a major contractor using in the Ada language.

Currently there are several Ada tasks in progress on this project. System engineers/software engineers and programmers are becoming comfortable with the language, and most importantly, Ada has given this project a Commonality among our major systems. In the past, one system used TACPOL, as many as four others used SIR (Symbolic Interpreter Routines), others used FORTRAN, and one used Ada. Now, this Ada commonalty among systems had system engineers discussing design approach, organization of Computer Software Components (CSC's), and interface problems with resident operating systems. Software engineers are discussing multitasking techniques, internal system requirements, system-to-system communications, and screen displays (formats etc.). Programmers are discussing different techniques to accomplish similar tasks and the reusability of Ada code between different systems. Most everyone seems enthused about learning more about Ada and enjoying it!

SOFTWARE EVALUATION TOOLS

The QA personnel maintain a large file of records. These records are used to locate and track areas of concern. To locate these areas certain evaluation tools must be applied. These tools are ADAMAT (Ada Measurement and Analysis Tool) and Trend Analysis (measure of changes in documentation, code, and test procedures, defect trends and complexity of code). These tools will be explained in part throughout this paper.

ADAMAT access has been given to all software individuals. Anytime they care to evaluate their source code, it is only a few keystrokes away. To allow comprehensive use of this tool the Quality Assurance Element created an ADAMAT Quick Reference Guide and the PM directed wide dissemination. Normally software personnel are not pleased with scores below 80%. (Programmers take great pride in their work.) Programmers have been observed many times studying their source code to raise their scores. Software personnel can also come to the QA office and review the documentation, call QA, or just send the source code for assistance with the analysis. The ADAMAT documentation is maintained in the QA office. This project places emphasis on reliability, maintainability, and portability of Ada. Comments are extremely important to the maintainability and portability of Ada. However, because this project requires the use of Program Management Shell (PMS) for each module and ADAMAT does not account for this in its evaluation, Self Descriptiveness is basically ignored for the automated evaluation of Ada. This project requires adequate commenting within Ada source code. This is verified during reviews and audits of the Software Development Files (SDFs). Ada source code is filed in the SDFs.

The QA specialist assigned to a specific system will evaluate the Ada source code a minimum of four times during the developmental cycle. The four evaluations are:

First. As soon as the programmer has completed unit testing on a Computer Software Unit (CSU) and the Software Engineer (SWE) has certified the unit. (we do not allow a programmer to certify their own code.)

Second. As soon as the SWE has signed the certification log in the CSC SDF stating the complete CSC has passed both unit and CSC testing. (Of course, if a CSC is comprised of a single CSU, then the first evaluation is eliminated.)

Third. After the first configuration management build prior to the Test Readiness Review (TRR).

Fourth. After the last configuration management build prior to system delivery.

The reasons for these four evaluations are as follows:

During the first evaluation QA will record the line count. This line count actually consists of three separate counts. Lines - total lines including blank lines. Statements - number of terminating semicolons, and comments - number of lines starting with (--). These line counts are recorded, the ADAMAT output report is printed, necessary files downloaded for the Metrics Display Tool (MDT) and all "Bad Coding Practices" recorded and filed in an active file.

The second evaluation will compare the line counts for changes, the output report printed, MDT files downloaded to the PC and "bad coding practices" recorded and filed.

During the third evaluation only CSC's are evaluated and compared. If any change is detected the CSU's are then run and also evaluated. Detected changes will be analyzed to determine the "Why?". These reports will be given to the applicable task manager.

The fourth evaluation is basically a duplication of the third evaluation. Any software changes made during formal testing are evaluated and documented. Upon completion of the fourth evaluation, all changes from initial certification to system delivery are documented.

QA will then take this information and compare it to other systems to determine shortcomings and areas of concern. If only one system shows a specific weakness in programming, a report is generated for that specific system. If multiple systems have the same specific weakness, a report recommending possible training will be sent to the Project Engineering Manager (PEM).

The previous paragraph began describing Trend Analysis. ADAMAT is a means of determining concern, and Trend analysis is a way to determine the seriousness of the concern.

Trend analysis is used to locate, document, and determine corrective action to prevent concerns from recurring.

COMMON CONCERNS

A sample of concerns detected, reported, and corrected is listed as follows:

1. Declarations. Use of a variable rather than a constant caused accidental change of an invariant. This occurred only a single time; however, reviewing Ada source for other occurrences showed this was a common mistake. Corrective action: System engineer notified and during the next evaluation, minimal occurrences were noted.

2. Functional Decomposition. A substantial number of instances occurred where large subprograms contained multiple functions and should have been broken into multiple modules. Detection of decomposition is basically a personal opinion. The Ada Style Handbook suggests 100 statements per module as a guideline. Normally, SWE's support the small, clear, concise module concept. This is quite common of programmers with less than one year experience.

3. Trial and Error Programming. This is using the compiler to verify syntax or determining the correct solution using a heuristic approach. (Granted this is one of the purposes of a compiler.) This can only be detected by close monitoring of compiler use. Once detected, a determination why this approach is being used in lieu of scientific principles must be made.

4. Arrangement of Compilation Order. During the beginning of the preliminary design, it was quickly determined that the order of compilation would cause long compile times, even if small amounts of code would change with some CSU's. The creation of subunits using "separate" became a engineering issue. Example. A date time group is placed on all configuration management builds and is displayed during the power on sequence. The location of this specific module can be critical to compilation times.

CHRONOLOGY OF EFFORT

SYSTEM REQUIREMENTS ANALYSIS

Requirements. During the system requirement analysis a Requirements Summary List (RSL) containing the general list of software requirements is to be implemented. Once a clear understanding of the RSL is reached, a Requirements Definition Document (RDD) is written. The RDD will carry each requirement from the RSL through the Preliminary and Critical Design increasingly explaining the requirements in more detail. A draft Software Development Plan (SDP) (DOD-STD-2167A requirement) is written and staffed within the task group for comments to insure continuity, and the SDP is submitted to the customer 30 days prior to the System Design Review (SDR). The start of the preliminary Software Requirements Specification (SRS) is to begin developing the requirements from general statements to very specific requirements. This document will also include responsibilities for the training and testing of each requirement. Rapid prototyping will also be conducted during this phase. Prototyping is used to assist in the early definition of screen displays and assist in the interpretation of difficult requirements. Interface specifications must be addressed in this software phase to define the exact data items both transmitted to and received from required devices. This document will be discussed at the SDR and delivered in final draft form for the Critical Design Review (CDR). Also the draft test plan (describing equipment needs and how the requirements are to be tested) will be delivered 30 days prior to the SDR.

Quality Assurance Approach: The QA specialist actively participates in the development and delivery of all documentation. This involvement ensures all documents conform to both the contract and to requirements listed within the DOD-STD-2167A. A continuing audit of requirements between the RSL and the preliminary SRS, between the RSL and the draft test plan, and the requirements between the RSL, the SRS, and the interface documentation is accomplished. The RSL will be subjected to Human Factor Engineering to insure that the requirements implemented in the Man-machine interface are user friendly. The QA specialist attendance at all working group meetings and formal reviews is considered essential.

Required Evaluation: QA will conduct both formal and informal evaluations during the software development phase. Some evaluations remain internal to the company while others are presented to the customer.

Internal: QA audits the RSL in comparison with the initial tasking document to insure all requirements have been transferred and to determine if new requirements have been added so proper tracking and traceability can be maintained.

External: QA will make a presentation of the evaluation criteria listed in Appendix D to DOD-STD-2167A. This appendix includes, "Traceability to the indicated documents", appropriate analysis", "design or coding technique used", "appropriate allocation of sizing and timing resources", and "adequate test coverage of requirements". Selected portions of this criteria are compared to the documents required by the customer. A copy of the evaluation is attached to the formal minutes of the SDR.

Other Necessary Evaluations: This phase imposes other necessary evaluations. These include the necessary equipment requests, monitoring and reporting of internal training classes, working group meetings, and format of documentation.

Ada unique features: Prototyping must be completed using Ada. Shortcuts are necessary and taken; however, Ada will be used. The SDP will contain design standards for both the Ada Design Language (ADL) and the Ada source code. QA will insure these standards are in accordance with both contract and military standards. The SDP development is also monitored for structured compilation orders and functionally organized modules with each CSC.

SOFTWARE REQUIREMENTS ANALYSIS

Requirements: During the software requirements analysis phase, both the SRS and the Interface documentation are delivered in final draft form. Both of these documents will remain in final draft until delivery of the software. This prevents unnecessary Engineering Change Proposals (ECP's) through the preliminary/detailed and formal test phases. Redlines are closely monitored and copies are distributed to all necessary agencies for comments.

Quality Assurance Approach: Traceability shall be carefully checked between the updated (if necessary) RSL and the SRS. All requirements, including size and timing constraints, specific responsibilities for training, and the testing of each requirement will be clearly defined.

Required External Evaluation: The formal evaluation presented at the Software Specification Review (SSR) will be similar to the formal review presented at the SDR. The documents are the final drafts of the SRS and the interface specifications.

Internal Evaluation: The SRS and interface documents shall be in the prescribed format.

Ada unique features: The structured approach in the SRS contains project unique identifiers which will also become the same names of CSC's during the preliminary and detailed design. This project unique identifier will assist in the traceability of requirements to modules and also assist in the design of some CSU's. This direct approach allows test case and test procedure development to begin at a much earlier part of the developmental cycle. Also because of the unique identifiers, some known algorithms may be coded this early in the cycle. Ada source code within the repositories may be researched to determine if similar applications exist. A PMS is developed to act as a permanent header for all developed source code and will be used for evaluation of reusable code at future dates. This PMS will contain version number, revision number and date, specification paragraph numbers for the SRS, IRS, and design documents. Also audit trail information is available; engineering number, Software Control Order (SCO) number, project unique identifier name, a description of the module, list of imported modules, list of called modules, instantiation, parameters, declarations, test cases (unit testing), and parent unit (if Separate) is used.

PRELIMINARY DESIGN

Requirements: The Software Design Document (SDD), the Interface Design Document (IDD), and the Software Test Plan (STP) are the documents required during this developmental phase. The test cases for all CSCs shall be identified and the CSC shall become the shell for the preliminary design. SDF's shall be opened for each CSU.

QA Approach: A close review of the SDD and the IDD back to the RSL and SRS to insure traceability is accomplished. Because a requirement may require multiple CSC's and requirements may share CSC's, traceability from each requirement to its CSC may be a difficult task. This traceability may be eased if the software task group will create a matrix listing detailed mapping of each requirement to its CSC(s). The traceability from the RSL and SRS to the STP for test identification is accomplished by a matrix in appendix I to the test plan. This matrix indicates each test case (identification) and the direct association to a requirement.

Required Evaluations: During the Preliminary Design Review (PDR), the evaluation criteria is the same as for all other reviews. The documents evaluated are the preliminary design of the SDD, the preliminary design of the IDD, the test cases established in the updated STP and the CSC test requirements for both CSC testing (programmer level) and identification of test cases (system testing).

Internal: The preliminary design must conform with the means described within the SDP. Announced and unannounced reviews and audits are performed to determine compliance.

External: A formal presentation will be made to the customer during the PDR. This presentation will cover the evaluation of the required documentation to the criteria established in Appendix D, DOD-STD-2167A.

Ada unique features: The preliminary design must contain compilable ADL. The CSC names must be identified the same as the project identifiers for traceability purposes. Coding accomplished during this phase must conform with the standards established within the SDP.

DETAILED DESIGN

Requirements: The required documentation for this software development phase is the updating of all draft documents. The SDD and the IDD will be updated to include the detailed design. The software development folders are updated to include CSC testing and perhaps, some test results available. The CSU's and the test case requirements have been identified. The STP will be updated with the individual test descriptions.

QA Approach: QA will conduct a detailed audit of each CSU and its associated CSC to assure the requirements are being fully implemented. Monitoring of general activities (development of code, SDF's, test cases, and procedures) and continuous. Review of all activities and completion of reports will also continue on a daily basis.

Required Evaluation: Evaluations of the updated SDD and IDD including their detailed designs and evaluation of each CSU (module) and their associated test cases toward the detailed design is accomplished. The CSC test cases for the test identification shall be evaluated to parallel the detailed design. Requirements and interface testing shall be evaluated for design completeness.

Internal: QA will monitor and review all SDF's. Insure the programmer notes are completed. Verify the detailed design contains a compilation order (CSC/CSU) designed to prevent long periods of compile times.

External: A formal presentation is completed at the CDR on the evaluation criteria in appendix D, (DOD-STD-2167A). The items evaluated are the updated SDD and IDD. CSU test requirements, test cases, and CSC test cases are evaluated for adequate detail. Inputs must be specified and expected outputs are mandatory. The SDP details how the SDF's will be maintained. Evaluation of the SDF's will be presented.

Ada unique features: Ada, when designed properly, the CSC/CSU use both the "WITH" and "SEPARATE" statement, and keeps compilation times to a minimum when minor modifications are applied.

CODING AND UNIT TESTING

Requirements: The QA will monitor the development of code to the schedule and with compliance to the SDP. Any deviation from the SDP will be documented by the programmer in the SDR. CSC and unit testing will be completed during this phase. All test procedures will be validated by interim system builds as full requirements are implemented. All software and test procedures will be validated prior to the Test Readiness Review (TRR).

QA Approach: QA will monitor, audit, review, and report the development of code as to the SDP. Prepare deviations from the SDP on a case-by-case basis. CSU testing will be closely monitored and results placed in the SDF. Monitor, audit, review, and report on the development of test procedures. Monitor all system builds and prepare trend analysis of similar problems encountered for each build.

Required External Evaluations: A formal evaluation will be presented at the TRR. This evaluation will include the source code for each CSU, CSC test procedures, CSU test procedures and test results, and an in-depth look at the SDF's.

Internal Evaluation: QA will evaluate the update of documentation and review the source code to MIL-STD-1815A and the SDP. The System Test Procedures will be validated. Validated means they are syntactically correct and have been run using interim software. SDF's are continually monitored to determine which CSU's and CSC's have been certified so the ADAMAT evaluation my be completed. Techniques are monitored and reviewed for compliance with the SDP and military standards/guidance. As test procedures are validated, some problems may be encountered with either the test procedures or the software. System Fault Reports (SFR) are initiated, analyzed, and the error corrected and verified. SFR's are maintained for input to trend analysis.

Ada unique features: The programmers can document in the SDF their different techniques applied to their code. And, if followed up, techniques are documented as to their own efficiency values. As a CSU or CSC is validated by the assigned SWE, the source code will be run through ADAMAT and the SWE given the output for evaluation. Compilation times are maintained and evaluated by the software task group for design considerations.

CSCI TESTING

Requirements: All CSC/CSU test results will be completed and filed in the SDF's. The test descriptions will be redlined during formal testing and the redlines incorporated into the final delivery. Source code and/or documentation will be updated to create the best software and associated documentation possible.

Quality Assurance Approach: All of the SDF's will be audited for accuracy and completeness. Formal testing is closely monitored and reviewed. All software/documentation changes made will be tracked to monitor reasons and justification. All changes will be applied toward trend analysis.

Required Evaluations: The SDF's will be evaluated at the completion of formal testing. Hardware and software concerns will be documented and reviewed at Data Review Boards (DRB's). The severity and status of all concerns determined at DRB's will be applied toward trend analysis. Any change to software or its associated documentation will be monitored, reviewed and reported.

Ada unique features: The modularity concept of Ada allows anomalies to be quickly located and easily changed. Turnaround time of changed Ada code is very quick because only modified CSU's (in most cases) must be recompiled and linked.

PORTABILITY, MAINTAINABILITY, AND REUSABILITY

The software task groups have the inherent responsibility for portability and maintainability of Ada software. During the software development cycle, close monitoring of each functional area is accomplished. QA monitors development of code in relationship to functional areas/groups for code duplication or assist in monitoring for reusability. QA also used ADAMAT to evaluate portability and maintainability. This monitoring not only occurs within each software development group, but also project wide. Because we have indicated specific responsibilities for Ada source code, these different agencies (Task groups, QA, Project Interoperability Coordinator, PEM, and the PM) strive in a teamwork manner to discuss and develop the best possible Ada source code.

CONCLUDING OBSERVATIONS

QA responsibilities are greatly enhanced for governmental Ada projects compared with non Ada projects. The Appendix D, DOD-STD-2167A, is very comprehensive for software and documentation evaluations. As a direct result of these evaluations, software and documentation are sometimes delayed but concerns can be detected and resolved much earlier in the software developmental cycle. Designing in Ada allows earlier programmer oriented testing (unit testing) and earlier development of test cases and procedures. Some Ada modules can be used from system to system and coding time can be saved.

CONCLUSION

QA responsibilities have significantly grown because of the new requirements imposed by DOD-STD-2167A and use of the programming language Ada. Continual efforts to learn, train, and apply new methods are critical to both QA and the software task groups to effectively work together and strive toward manufacturing the best Ada source code and associated documentation possible.

For software producing companies to remain competitive, reusability, portability and maintainability of Ada source code is critical.

REFERENCES

Department of Defense Standards

1. DOD-STD-480A Configuration Control - Engineering Changes, Deviations and Waivers.

2. DOD-STD-1467 Software Support Environment dated 18 January 1985.

3. DOD-STD-1679A Software Development dated 23 October 1983.

4. DOD-STD-2167A Defense System Software Development dated 29 April 1988.

5. DOD-STD-2168 Defense System Software Quality Program dated 29 April 1989.

Military Standards

1. MIL-STD-1521B, Technical Reviews and Audits for Systems, Equipments, and Computer Programs dated 4 June 1985.

2. MIL-STD-483A, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs dated 4 June 1985.

3. MIL-STD-490A, Specification Practices dated 4 June 1985.

4. MIL-STD-1815A, Ada Programming Language dated 22 January 1983.

MISCELLANEOUS

1. DOD-HDBK-287 Military Handbook, Defense System Software Development Handbook dated 8 August 1986.

2. AFCS PAMPHLET 800-43 Software Management Indicators dated 31 January 1986.

DAVID DISBROW

TELOS Systems Group
P. O. Box 33099
Ft Sill, Oklahoma 73503-0099

DAVID DISBROW is currently a system engineer for the United States Marine Corps FIREFLEX system working with the Center for Software Engineering, Fire Support Systems, Fort Sill, OK. He is employed by TELOS and has significant experience as a quality assurance specialist and system test specialist. David has a B.S. in Electrical Engineering and is working toward his masters in computer science. His interests include software development methods, analysis of design methods and the Ada programming language.

DAVID MARTIN

TELOS Systems Group
P. O. Box 33099
Ft Sill, Oklahoma 73503-0099

DAVID MARTIN has an Associate Degree in Computer Programming, a Bachelors Degree in Technology, and has been accepted into a MBA program. He has been a Quality Assurance Specialist for six years and has worked with DOD-STD-2167A projects for the last two and one-half years.

A LOOK AT SEI SOFTWARE PROCESS ASSESSMENTS

Terry B. Bollinger and Clem McGowan

Contel Technology Center
Chantilly, Virginia

Abstract: In recent years, the concept of *process assessments* has received much attention in the software community. A process assessment is an analysis of how parts of a software project (e.g., people, tasks, standards, and resources) interact to produce software. This paper analyzes a form of process assessment that is advocated by the DoD-sponsored Software Engineering Institute (SEI). The authors conclude that while the SEI has developed a truly outstanding program for performing process assessments, its parallel effort to rank software organizations in terms of a five-level "maturity framework" is seriously flawed in its reliance on an unproven process model and its sparse-coverage approach to collecting and analyzing process maturity data. The paper ends by introducing a new, more quantitative approach to analyzing and optimizing processes.

Introduction

In recent years, the concept of *software process assessment* has become increasingly important both to the software community in general, and to the DoD software community in particular. In the most general usage of the phrase, a software process assessment is simply an evaluation of how the many parts of a project — people, tasks, tools, standards, and various types of resources — interact to produce software. The central objective of a process assessment is to understand and improve the process by which an organization builds high-quality, high-reliability software products.

Software process assessments are of particular interest to companies that develop software for the government because of a program at the Software Engineering Institute (SEI), which is a DoD-sponsored research and development center in Pittsburgh, Pennsylvania. SEI's Software Process Program, which is headed by Watts S. Humphrey, began as an effort by the U.S. Department of Defense to find a more quantitative way to evaluate the capabilities of software development organizations.[1] Such information would then be used to "weed out" low-scoring organizations during bidding for DoD contracts. In this fashion the DoD hopes both to avoid expensive overruns due to inexperienced software contractors, and to provide a rather powerful incentive for low-scoring organizations to improve their software process.

Although their process program originated as a approach to auditing software organizations, SEI quickly realized that certain aspects of their program had good potential as a way of transferring software and process technology. The SEI effort therefore split into two closely intertwined (but distinct) programs, as shown in Figure 1.

Thread 1: Capability Evaluations

Purpose: Help DoD select, monitor contractors

- DoD performs evaluation, knows all results
- Can significantly influence contract awards
- May be used to "motivate" contractor changes

Thread 2: Process Assessments

Purpose: Assess current software practices

- For internal, confidential use of contractors
- Intended to help "unfreeze" current processes
- Provide inputs for a self-improvement plan

Figure 1 - Evaluations Vs. Assessments

The second, newer thread of the SEI Software Process Program is *process assessment*, which is focused on technology transfer, self-help, and non-attribution of results. Despite the fact that they share many of the same materials and key concepts, the contrasts between SEI assessments and SEI evaluations are quite marked.

One analogy sometimes heard is that an assessment is like hiring a tax consultant to help prepare your taxes, while an evaluation is like having the IRS audit your taxes. The group dynamics are, shall we say, a bit different in the two cases. SEI assessments encourage a surprisingly free and open exchange of information between members of an organization, while evaluations (audits) are more likely to produce the opposite reaction: great reluctance to answer or discuss anything except the exact questions posed by the auditors. As a result, process assessments tend to be better than evaluations at gleaning detailed insights into how a given software processes operates.

Goals of This Paper

The authors of this paper have participated both in SEI training classes, and in a number of modified (non-SEI) process assessments within their own organization. As a result, they are firmly convinced that process assessments are an excellent tool to improve the overall productivity and quality of software organizations, and that the SEI model for performing such assessments should be viewed as a major contribution to the software industry. However, process assessment is only one of two threads in the SEI Software Process Program. Understanding how the other thread of capability evaluation works requires a much closer look at how information is collected for an evaluation, how that information is analyzed, and the model (SEI's "process maturity framework") by which organizations are ranked.

In keeping with the spirit of SEI process assessments, this paper will attempt to look more closely at how SEI currently performs assessments and evaluations, with a particular emphasis on their evaluation program. It is the hope of the authors that this feedback can then be used by SEI to help examine and update key features of their own Software Process Program.

Specific issues that will be discussed and analyzed in this paper, although not necessarily in the following order, include:

- **SEI Process Assessments.** How well do the non-graded, non-attributed SEI process assessments work? How do people within an organization respond to such assessments? Is confidentiality important? Are there any specific ways in which the current SEI assessment program could be improved?

- **The SEI Process Maturity Framework.** In both SEI assessments and evaluations, the "process maturity" of an organization is rated in terms of a simple five-level model called the Process Maturity Framework. Are the definitions these five levels reasonable and appropriate? Is it based on the structure of software organizations that have proven themselves effective at producing quality software, or is it hypothetical? Does it represent a consensus view of the software community? Has it ever been thoroughly tested?

- **The SEI Process Maturity Questionnaire.** In both SEI assessments and evaluations, the ranking of an organization in the one of the five Framework levels is based on the results of 85 bits (about 11 bytes) of yes/no information extracted from a 101 question SEI Process Maturity Questionnaire. Are the questions in this form reasonable, unambiguous, and sufficiently general to apply to a broad range of DoD software needs? Are they fair? Is the grading system by which the questions are analyzed equitable? Does the questionnaire make "cheating" difficult? What does the questionnaire imply that an organization must do to "move up" to a higher level (grade) in the SEI maturity framework? How does it handle technology issues?

- **Fundamental concepts.** The SEI process model is explicitly based on the assumption that the quality control concepts used to manufacture assembly-line products such as cars, cameras, wristwatches, and copper sheeting can be directly applied to software.[1,2] Is this assumption really valid for the entire range of software products developed by the DoD, or is it an oversimplification that applies only to a quite narrow subset of the overall DoD software development and software maintenance problem?

SEI Process Assessments

The major steps in an SEI process assessment are shown in Figure 2. One of the interesting and significant features of an SEI assessment is that it always begins "top down," with high-level management signing up to the program before any further actions are taken. This approach provides a mechanism by which organization-wide change can be accomplished, and also gives participants from the organization more confidence that their inputs may result in real actual changes within the organization.

After high-level management "buys into" an assessment, SEI provides intensive training to a carefully selected team of people from the organization. Once trained, this team assumes the role of internal consultants who are responsible for actually performing the assessment. The trained team prepares for the assessment and selects specific projects and functional groups to be interviewed. Prior to the actual assessment, the selected projects and groups are asked to fill out a questionnaire about their current software process.

Steps in an SEI Process Assessment (1990)

1. **Sign up.** High-level management agrees to an assessment, and commits to improvement

2. **Training.** A small team from the organization are trained in how to perform assessments.

3. **Preparation.** The trained team prepares to insure an orderly one-week assessment

4. **Questionnaire.** Selected projects and groups fill out SEI questionnaire (101 yes/no answers)

5. **Assessment.** The trained team performs an intensive week of project and group interviews

6. **Presentation.** The week concludes with the presentation of key findings to management

7. **Report.** The trained team produces a report on key findings and how to implement them

8. **Implementation.** Management works to follow through on improvement commitments

Figure 2 - Steps in an SEI Process Assessment

The actual on-site assessment usually takes from three days to a week, and consists of an intensive sequence of interviews and guided group discussions. Due in large part to SEI's excellent preparation of the assessment teams, these interviews and discussions are usually surprisingly positive in both tone and content. Team leaders are trained to encourage an atmosphere of open discussion without fear of repercussions, an atmosphere that is possible only because of the very strong emphasis SEI places on non-attribution of all assessment data. Final recommendations are devised based on a group consensus by team leaders, project representatives, and functional group representatives, and are presented to management on the last day of the assessment.

Finally, the assessment team is responsible for writing a report that defines specific strategies for improving the software processes of the organization. The chances that the report will actually be implemented — a concern familiar to many software engineering and quality assurance groups — is greatly increased by the fact that high-level management has explicitly commissioned the assessment and thus committed itself publicly to taking action on the resulting report. While this arrangement does not guarantee implementation, it at least ensures that the report cannot be as easily forgotten or put aside.

Although the SEI assessment program shares many of the same materials as the SEI evaluation program, its reliance on those materials is much less than one might suppose. What makes the SEI assessment program truly remarkable is not its "model" for software maturity, but is its ability to select people from projects and groups that may not even be on speaking terms and weld them together into a forward-looking, process-oriented team.

For example, a common result of an assessment is that groups with poor relations come to understand how their conflicts are often just symptoms of a poorly structured processes, and not just "personality conflicts." This type of information can be very valuable, since it permits long-standing conflicts to be converted into specific insights on how to construct a more effective process.

Based on an analysis of both of the SEI training materials and experience with actually applying the SEI process assessment model, the SEI Process Assessment program appears to be both well-formulated and highly responsive to feedback from organizations that have used it. Its strict emphasis on confidentiality and non-attribution of results are remarkably effective at helping to elicit inputs and feedback that could never otherwise be obtained. Overall, the SEI process assessment program is a powerful "people tool" that has been finely tuned to work well with many groups and organizations, and which consistently produces valuable results for organizations that follow it carefully and work to use it.

The only significant area in which the SEI process assessment may be lacking is found in the fact that it elicits far more information about processes than it actually succeeds in recording. The final report provides major recommendations, but does not provide details about the structure and functions of the software process or processes that currently exist in the company. Such information is quite valuable, since it provides a specific framework for making follow-up recommendations on how to improve the organizations software processes, and also provides a starting point for new projects that have not yet defined their software processes.

SEI Capability Evaluations

Procedurally, an SEI capability evaluation resembles a simplified form of a process assessment. The team that evaluates the organization is from the government, so there is no need for team training. Interestingly, the same questionnaire is used in process assessments and capability evaluations — but the implications of filling out the questionnaire are very different in an evaluation. Rather than being just the starting point for detailed on-site discussions about the process, in an evaluation the questionnaire becomes the focal point for grading and ranking the organization. Discussions between the evaluation team (auditors) and the organization thus tend to focus on verification of the answers that were given on the questionnaire, rather than identification of specific process improvement issues and concerns. For example, if an organization claims that they have coding standards that are applied to all projects, they may be asked to provide examples of both the coding standards and code from all projects in which those standards have been used. (Note that the IRS audit analogy is quite apt.) Recommendations reports are written for evaluations, although organizations tend to view these reports as explanations of why they failed to receive a higher rating.

Because of the audit-style arrangement of evaluations, the key to understanding how they work is to understand how they actually acquire and process audit information. There are, of course, many other issues involved (e.g., how the auditors should go about verifying that a group actually does what it claims), but unless the underlying set of SEI grading scales and grading methods are fair to begin with, these logistical aspects of SEI evaluations are only secondary concerns. In this SEI assessments and evaluations differ vastly, for in a consensus-building assessment the *way* in which the assessment is done is absolutely vital, while in an evaluation the "group dynamics" are of far less importance in comparison to how the final one-to-five "grade" is derived.

The two components of SEI ranking and grading are:

- <u>SEI Process Maturity Framework.</u> This is a five-level scale used to rank the "process maturity" of an organization, and to provide guidelines for how an organization can increase its maturity.

- <u>SEI Process Maturity Questionnaire</u>. This is a set of 101 yes/no questions that must be answered by the organization prior to an evaluation.

What these two components are and how they work are described in the next few sections. Several aspects of the two components are then analyzed in greater detail.

The SEI Process Maturity Framework

This five-level scale is described in detail in Watts Humphrey's book *Managing the Software Process.*[2] However, the "authoritative" definition of the levels must be viewed as being the one embedded in the SEI Process Maturity Questionnaire, since it is the questionnaire and not the book that is used to actually grade an organization. Figure 3 briefly describes the five levels of the SEI Process Maturity Framework.

SEI Process Maturity Framework (1990)

The SEI Process Maturity Framework is a five-level scale used to "grade" software organizations, and to provide specific goals for process improvement. Data for rating an organization is taken from the 85 graded yes/no questions (out of a total of 101) in the SEI questionnaire.

LEVEL 1 — "Initial" Processes

- SEI: *"Unpredictable & poorly controlled"*
- Includes nearly all "unprepared" organizations
- *No* entry requirements, so skills may vary greatly

LEVEL 2 — "Repeatable" Processes

- SEI: *"Can repeat previously mastered tasks"*
- Focus on management and tight project control
- Focus on collecting various types of "trend" data

LEVEL 3 — "Defined" Processes

- SEI: *"Process characterized, fairly well understood"*
- Focus on software design skills, design tracking
- Focus on various types of traceability

LEVEL 4 — "Managed" Processes

- SEI: *"Process measured and controlled"*
- Focus on technology management and insertion
- Focus on estimates/actuals, error cause analysis

LEVEL 5 — "Optimizing" Processes

- SEI: *"Focus on process improvement"*
- Focus on rapid technology updating, replacement
- Focus on process optimization to reduce errors

Figure 3 - The SEI Process Maturity Model

The SEI framework will be discussed throughout the paper, but a few features worth noting here include:

<u>Level 1 really means "Failed."</u> Since there are *no* requirements for achieving Level 1, it is really best viewed as a catchall category for all organizations that failed to meet the minimum requirements of

Level 2. It therefore includes a much broader range of organizational skills than the other levels, ranging from organizations totally incapable of producing software to ones whose actual track record for producing quality software on time may be very good.

- **Management controls are emphasized.** The SEI framework places an "up front" emphasis on getting a process under management control. Management control is the major theme of Level 2, and all higher levels are assumed to have such controls in place before other techniques and issues are added.

- **Measurement is a consistent theme.** Levels 2 through 5 of the framework all emphasize product measurements and the use of such measurements.

- **Process optimization is left till last.** It is interesting to note that process optimization is not addressed until Level 5 of the process framework. The implicit (and highly debatable) message of this ordering is that process optimization is not effective until the process is very tightly controlled and measured.

Many of the above features tend to reflect a fundamental precept of the SEI framework, which is that software processes are fundamentally quite similar to the types of assembly-line processes by which cars, watches, and copper sheets are made. Recognizing the influence of the assembly-line paradigm helps explain some of the odd features of the SEI framework, such as its delay of process optimization until each "part" of a software product has a well-defined "assembly line" that can itself be measured and improved. In practice, however, it is quite rare for the software products of an organization to be so remarkably uniform in function and form that the software process used to develop them will *ever* "evolve" into a fixed set of "assembly lines" for building individual "parts" of those products. Such issues of generality are simply not addressed by the overall SEI framework.

The SEI Questionnaire

The SEI Process Maturity Questionnaire consists of 101 yes/no questions on a variety of software engineering and process issues (see Figure 4). One rather surprising feature of the questionnaire is that it effectively ignores tool and technology issues, even though 16 questions on the test are devoted to that subject. The SEI grading scheme ignores the technology section.

It should be noted that for some time now a major effort to update and expand the SEI questionnaire has been underway, and that a new SEI questionnaire with more questions and broader coverage should be released sometime in 1991. However, the results of that work have not yet been made public. It is hoped by the authors that the new questionnaire will deal with at least some of the issues discussed below. However, until the new form of the questionnaire is released, the current (1990) version is still the official document against which software organizations can and have been evaluated, and the results derived from this form can and have been factors in the consideration of contract awards by DoD.

SEI Process Maturity Questionnaire (1990)

The current (1990) Process Maturity Questionnaire is identical to the first version that was released in the late 1980s. However, it is now undergoing extensive review and a significantly updated version should be released sometime in 1991.

There are a total of 101 yes/no questions. However, the Tools and Technology section (16 questions) is ignored by the SEI grading process, so only 85 questions are actually used during an evaluation.

Organization and Resource Management (17 total)

- Organizational Structure (7)
- Resources, Personnel, and Training (5)
- Technology Management (5)

Software Engineering Process and Its Management (68 total)

- Documented Standards and Procedures (18)
- Process Metrics (19)
- Data Management and Analysis (9)
- Process Control (22)

Tools and Technology (16 total)

Figure 4 – The SEI Process Maturity Questionnaire

This philosophy of ignoring technology issues during a process evaluation is deeply embedded in the SEI model, and it shows no sign of abating in SEI's current approach to capability evaluations. For process assessments, in which the focus is on helping people improve how they work together, such a focus seems fairly plausible. However, for an SEI *evaluation*, in which the nominal goal is to ascertain just how effective and reliable an organization is at "bottom line" software production, such a de-emphasis on automation and tools seems seriously out of place.

What is worrisome about an evaluation process that ignores technology and tools is this: It stands a real risk of rewarding organizations for building very elaborate, very expensive manual processes to accomplish work that could in many cases be done by a fully automated tool. The history of the software industry is replete with examples where in which products such as operating systems, compilers, and databases made the manual processes that preceded them obsolete.

For example, imagine that a software group back in the early days of compilers had developed a very good but manually intensive process for developing mathematical aerospace software. At the same time, a competing group realized that a Fortran compiler could drastically cut the cost of developing such software and also greatly reduce the number of code defects due to mistakes in the manual translation of formulas into assembly code.

An evaluation model that ignores technology as a way of automating parts will tend to be blind to such issues, and could very well score the first group as having a "better" process — despite the fact that its hands-on approach is more like to produce erroneous products, and at a significantly higher cost! The importance of avoiding such people-over-automation decisions for processes that could be automated is discussed by B. Barnes and one of the authors in a recent article on software reuse.[3]

The key point in the above example is that the second group has not really eliminated any process steps — they have instead used available technology to *automate* the error-prone process step of converting formulas into assembly language. Thus the addition of a tool in a case like this clearly meets the spirit of the SEI program, since it moves the overall process towards greater reliability by increasing the rigor of how a key part of the process is done.

One counter argument might be that recognizing a good process is valuable in and of itself, and thus should not be confused with the "separate" issue of automation and tools. There is a very simple reason why this argument does not hold up well — it is that the DoD has used the results of SEI evaluations to help in contractor selection.

Contract awards are traditionally made on the basis of who is most likely to produce a high-quality product at low cost, *not* who has the best "process." And indeed, the materials for SEI assessments and evaluations state repeatedly that their reason for analyzing processes is to provide a better understanding of which organizations will produce higher quality software at a lower cost. However, given the very substantial skewing effect that process automation can have on an organization's effectiveness and ability to produce quality products, the assumption that process quality *alone* is a sound metric for comparing software organizations seems unrealistic.

A second counter argument might be that Level 5 of the SEI Process Maturity Model *does* emphasize technology and technology transition, but intentionally avoids making it into a key issue until the organization has extensive management and process controls in place. The rationale behind this approach is to keep groups from "floundering" when trying to use new methods.

One problem with this argument is that it ignores the fact that there may *already* be many organizations making quite effective use of new technologies to shorten or increase the efficiency of their software processes — after all, there are more than a few organizations out there that are using compilers, database systems, and tools quite effectively without ever having had to reach the equivalent of Level 5 in the SEI framework.

More importantly, one must also ask whether such technology transition issues really should be pushed off until Level 4 (a "managed" process) or Level 5 (an "optimizing" process) of the SEI model to begin with. Given that very few software organizations would at present rank above Level 1 or Level 2 in the SEI model, is it really true that such organizations should halt their acquisition of new tools and cease looking at ways to automate their own work until they reach Level 4 or Level 5 of the SEI maturity scale?

Questions of this type deal as much with the nature of the SEI grading system as they do with the SEI process maturity framework, so let us now take a closer look at that grading system. Additional information about the questionnaire (e.g., examples of actual questions) will also be presented as part of the discussion of the grading system and framework.

The SEI Process Maturity Grading System

As mentioned before, the SEI grading system is based on 85 yes/no questions that appear in the SEI Process Maturity Questionnaire. However, unless they have had specific SEI training in how to grade questionnaires, most people who fill out one of these forms will have no idea at all how the form will be graded. This is a bit odd in itself, since one of the nominal purposes of the questionnaire is to provide organizations with insights as to how they can improve their software processes.

The exact algorithm for grading the questionnaires can be derived from the pubic-domain grading templates, however, so any organization sufficiently motivated — say, by losing a big contract to a competitor with a higher ranking — can certainly uncover the exact SEI grading scheme and uncover its implications.

The SEI grading scheme is shown in Figure 5. It can be viewed in either of two ways: 1) for a scheme used to grade only 85 true/false questions, it is remarkably complicated, or 2) for a scheme that may determine the financial fate of an organization that works primarily on DoD contacts, it is remarkably simple. Both views have definite merit.

The best way to understand the SEI grading system is to think of it as a series of seven hurdles, each of which is really a very small true/false test on a specific set of questions. Given the quite small size of these sets, SEI has taken the view that each set represents a sampling of many possible issues and concerns for a particular level. However, the sampling is not a dynamic thing, at least is not in the current (1990) version — the contents of each set of questions have remained exactly the same since the questionnaire was first introduced in the 1980s.

(Interestingly, the main reason behind SEI freezing the questionnaire appears to have been that any attempt on their part to change it provoked a flurry of complaints from industry that the "rules" (the 85 graded SEI yes/no questions) for a "good process" (apparently meaning one that implements the 85 *questions*) were being "changed" (added or deleted) just as the organizations began building their software processes around them!)

Given the seven-hurdle structure, one would rather naturally tend to assume that it results in eight different maturity rankings. Not so. Three pairs of the failure branches are instead grouped to create Levels 1, 2, and 3, with single branch points defining Levels 4 and 5.

For such a small sampling space, this is a remarkably complex grading algorithm. For example, one simple grading algorithm that is effective and reasonably fair for such a small sampling space is percentage-style cutoffs such as those used in grade school and high school tests. The problem with trying to apply overly complex grading schemes to small data sets is that they tend to chop up the data into smaller and smaller "mini-tests" that are statistically very unreliable.

As a worst-case example of how a complex grading can lead to quirky results, imagine a test with 85 randomly selected questions in which *each question* is a mini-test "prerequisite" for some other question. The chances of failing at some point in such a test would be very high, since missing *any* question along the way would result in the taker of the test being "dumped out" at that point. Even worse, a very knowledgeable test taker who happened to miss one of the first few questions would receive a very low score — even if she or he answered *all* of the higher level questions correctly!

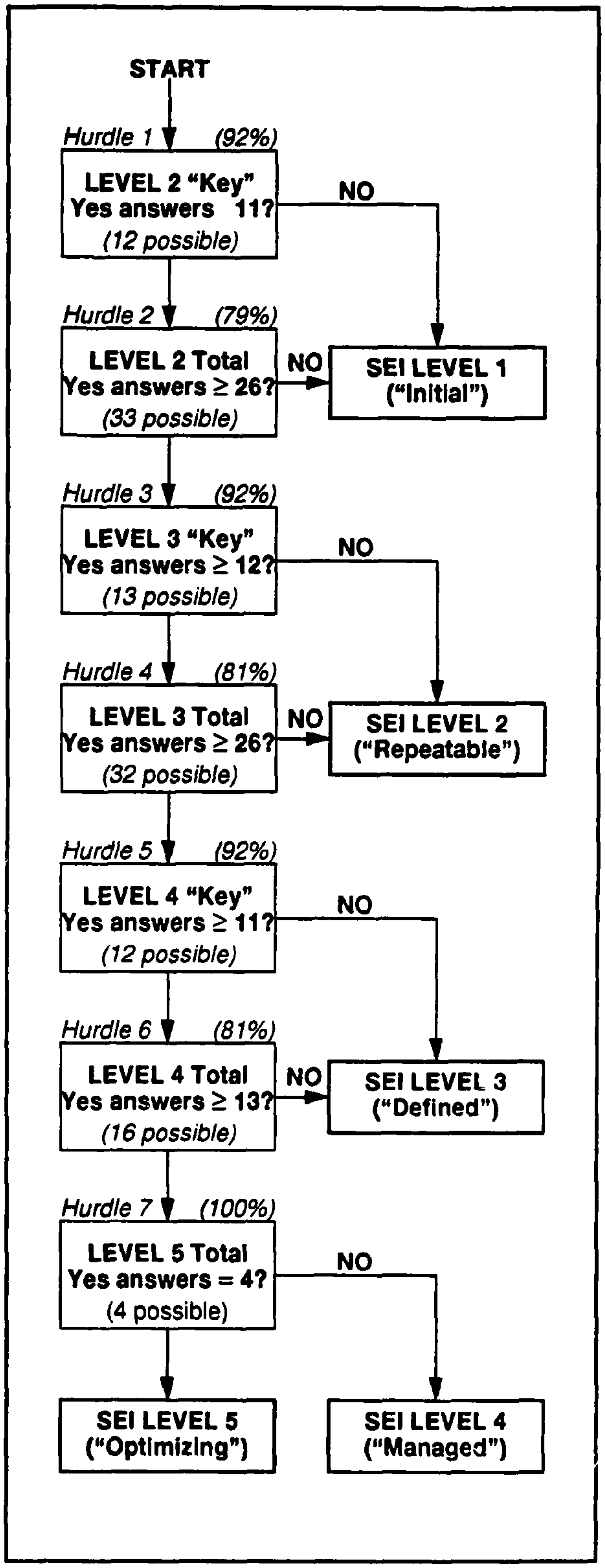


Figure 5 - How SEI Questionnaires are Graded

While the SEI grading system shown in Figure 5 is not as bad as the example just given, it is not that much better, either. As a worst-case example, an "unprepared" organization (that is, one with no foreknowledge about the SEI questionnaire or grading system) could answer yes to all but *two* of the total set of 85 graded questions, *and still be classified as an "unrated" Level 1.*

Another troublesome feature of using multi-hurdle grading algorithms in combination with small numbers of random questions is that it can introduce a false sense of progression that is an artifact of the *grading system,* rather than an inherent feature of the subject matter of the test. The two factors that join together to create such artificial progressions are:

1. the fact that a "chained" grading system can make a set of test questions much, much harder to pass than they would be if graded under a simpler system, and

2. the fact that those taking the test will tend to focus first on the earliest questions in the test chain, regardless of whether those questions are the easiest or most difficult in the subject area.

In the earlier example a grading system that "chained" all 85 questions, these two effects can combine to make a seemingly simple test on a familiar topic into a complex and confusing affair. If you add to it the possibility that the teacher *might not tell you* what the exact "grading sequence" is, such testing methods begin to resemble a nightmarish game of blackjack. Those being tested must undergo repeated failures just to figure out what cards are where in the "stacked" grading deck.

On the other hand, such grading methods can certainly induce furious preparation on the part of those being tested. The problem, though, is that the *order* in which they prepare may or may not have any real relationship to the subject area. That this is true can be seen simply by asking what happens if the order of the questions is *exactly inverted.* The answer, of course, is the same: it induces furious study on the part of those being tested, except that now they will study the most "difficult" subject first!

Although the seven hurdles of the SEI grading systems are less extreme than chaining all 85 questions, it is still complex enough to create a strong artificial progression effect. For example, it is quite common in SEI process assessments to hear people complain mildly that although they can honestly answer yes to many Level 3 questions, they cannot grade those answers because they previously missed two or more Level 2 Key Questions (see Figure 4). Such complaints are often a good indicator of an unfair grading system, and should not be dismissed lightly.

To understand what such effects mean for SEI testing, it is helpful to look more closely at what one of the hurdles in the SEI grading system looks like. The first SEI hurdle is the set of 12 questions quoted verbatim in Figure 6. (Italics in the questions are from the SEI questionnaire.)

SEI Level 2 "Key" Questions (1990)

1. Does the Software Quality Assurance (SQA) function have a management reporting channel separate from the software development project management? (SEI Question Number 1.1.3)

2. Is there a software configuration control function for each project that involves software development? (1.1.6)

3. Is a *formal process* used in the management review of each software development prior to making contractual commitments? (2.1.3)

4. Is a *formal procedure* used to make estimates of software size? (2.1.14)

5. Is a *formal procedure* used to produce software development schedules? (2.1.15)

6. Are *formal procedures* applied to estimating software development cost? (2.1.16)

7. Are profiles of software size maintained for each software configuration item, over time? (2.2.2)

8. Are statistics on software code and test errors gathered? (2.2.4)

9. Does senior management have a *mechanism* for the regular review of the status of software development projects? (2.4.1)

10. Do software development first-line managers sign off on their schedules and cost estimates? (2.4.7)

11. Is a *mechanism* used for controlling changes to the software requirements? (2.4.9)

12. Is a *mechanism* used for controlling changes to the code? (Who can make changes and under which circumstances?) (2.4.17)

Figure 6 – SEI "First Hurdle" Questions

Note that because of the critical placement of this hurdle at the very beginning of the SEI grading chain, missing *any two* of the 12 questions in Figure 6 means automatic failure (a Level 1 rating) of the *entire* SEI evaluation! The implication is that any organization that fails to pay very close attention to the implied recommendations of these

12 questions may easily find their competitive position for winning DoD contracts adversely affected.

The reason they must pay close attention is the questions in Figure 6 would be very difficult to guess ahead of time — even by an organization that is well-structured and would consider itself to be "mature." The problem here goes back to the earlier discussion of the dangers of testing based on small numbers of more-or-less random questions — because the number of data points is so low, the statistical reliability of the test becomes quite doubtful.

For example, an organization might already collect and use extensive data about their software process and products, such as the number and type of problems found in design and code walkthroughs, complexity measures of their software and the relationship of those measures to test results, and total work expended each time a change is implemented on a baselined interface definition. But if that organization happens to believe that the much simpler metric of *size trends* (Figure 6, Question 7) is neither relevant nor useful — well, they are then 50% of the way to failing their SEI evaluation!

A full failure could then come about in this way. Suppose that the organization was also very strongly committed to quality assurance, so that managers and developers who could not show a clear commitment to quality simply did not get hired. As a result, the organization developed an unusually solid and close working relationship between developers and quality assurance personnel, one in which developers viewed QA people as important members of their team who were responsible for finding problems before they mushroomed into extra work for the developers. Due to this unusually good working relationship, the organization developed a policy of hiring QA people directly into projects so that they could focus on the specific QA needs of each effort.

It would be a serious understatement to say that such an organization was in keeping with the spirit of process-level quality assurance — but how would it rate in terms of the first SEI hurdle? The answer, alas, is that none of the complex history, reasoning, or even actual quality results of such an organization would show up under the SEI testing scheme, because the organization would fail to meet the first question of Figure 6, which asks only whether QA people have a management chain available to them that is separate from that of the project. In this case, two strikes mean you are out, and the hypothetical organization would immediately receive a Level 1 rating, regardless of how they answered any other items on the SEI questionnaire.

Such a result is even more interesting if one considers what some of the SEI literature[2] has to say about the quality and dependability of Level 1 organizations. A Level 1 organization is characterized by ". . the lack of a managed, defined, planned, and disciplined process for developing software," [2] and it is also said that of Level 1 organizations that "[u]ntil the process is under statistical control, orderly progress in process improvement is not possible." [2] Such descriptions clearly would not help the hypothetical organization get new work, since anyone familiar with the SEI grading scale would tend to assume that the evaluation had "proved" that such descriptions were applicable.

One may of course respond that the above organization is only hypothetical and thus highly unrealistic, or that any organization that was that good could very easily reconfigure details of how it operates to keep itself in line with the SEI questionnaire. However, that would overlook the main point of the example. The main point is that when developing a testing strategy for analyzing a complex system, the use of a very sparse set of one-bit data points will always run a significant danger of giving a misinterpretation of the real situation. We will return to this theme of *sparse data analysis* later in the paper.

Despite all of the above points, it should be noted that the actual *subjects* covered by the first hurdle questions are both reasonable and appropriate. The odd grading possibilities tend to arise more as a result of using a very small number of narrowly focused questions to collect information, rather than from what subjects are covered in the hurdle. For example, the questions in the first hurdle can be grouped into four themes:

1. **Management controls.** Four questions (1, 3, 9, 10) deal with basic management controls and policies of the organization.

2. **Configuration controls.** Three questions (2, 11, 12) deal with configuration control at both the code and requirements levels.

3. **Project estimation.** Three questions (4, 5, 6) focus on "formal" procedures for estimating project-level issues such as sizes, times (schedules), and costs.

4. **Data collection.** The remaining questions (7, 8) deal with collection of data on code size and numbers of errors found during testing.

With the possible exception of the fourth theme of data collection, these are quite reasonable topics for a disorganized project or process to address. The themes of management control and configuration control seem particularly appropriate, since they provide an immediate payback of reducing confusion and lost effort in a project.

Project estimation also seems to be a reasonable theme to emphasize early on, since it complements the themes of near-term project and product control with better safeguards against unrealistic future commitments.

The last theme of data collection is more surprising, since it is not obvious what a supposedly immature software organization should *do* with a metric such as code size trends after collecting it. Possibly this is intended to get data collection off to an early start, so that the organization will have the data available for later use. The most likely interpretation, however, is that this is part of the overall theme of gaining "statistical control" over the software process, a theme that derives primarily from the SEI assumption that software development is closely related to assembly-line manufacturing.

The debatable nature of the data collection theme (a theme whose two questions could *by themselves* cause organization to fail an evaluation) underscores that the twelve first hurdle questions do not represent an industry or academic consensus about the twelve most important features of a new software process. That is not to say that the questions went without review before they were frozen into their current form. In fact, Watts Humphreys states that the questionnaire was reviewed by more than 400 government and industry organizations. However, there is no indication from SEI literature that the *grouping* of questions into the seven critically important grading hurdles was reviewed or even extensively discussed outside of SEI. Even if it had been, it is quite unlikely that the very diverse needs of government, aerospace, and business groups could ever be realistically packed into such a brief definition of what should come first.

As mentioned earlier, a more realistic interpretation of the seven sets of hurdle questions is that they are a sparse, more-or-less random selection of data points taken from the major themes of the SEI questionnaire (see Figure 4).

What About the Next-Generation Questionnaire?

One might respond to all of the above discussion about the 1990 SEI questionnaire by quite correctly pointing out that SEI is keenly aware that their current questionnaire has problems, and they have been working hard to correct them. (One might also note that these concerns have not kept SEI from actively using the current questionnaire to evaluate and rank companies in bidding situations, and that the current questionnaire thus must be viewed as considerably more than a "prototype.") Isn't it likely that the new questionnaire will correct most or all of the problems mentioned above, thus making all of this analysis irrelevant?

The new questionnaire almost certainly will represent a substantial improvement over the current one, since it should be based on several years of feedback from numerous SEI assessments and evaluations. Also, the current review process includes invited representatives from industry, so there is clearly a sincere SEI interest in getting industry feedback for the new questionnaire.

However, the authors are not aware of any indications from SEI that the basic format of the questionnaire — that is, a "checklist" of yes/no questions about various process and software engineering topics — is going to change significantly, or that the SEI Process Maturity Framework will be significantly revised. Furthermore, it is not clear whether the grading system is being carefully reviewed at the same time — and as we have seen in the earlier discussions of this paper, the grading system is very important in determining the fairness and statistical reliability of such a questionnaire.

A plausible guess is that the new SEI will be more flexible and will include a greatly expanded, more thoroughly reviewed set of questions. However, it probably will still be based on the same general concepts of random sampling with yes/no questions, a multi-hurdle grading system, and a five-level process maturity ranking scale. Even if it is used in a more flexible fashion, the total set of questions is likely to remain fixed, since questions of fairness would arise if entirely different sets of questions were used to rank organizations.

Can a framework such as the one just described be used to correct all of the problems identified so far with the SEI approach to evaluating software organizations? At the very least, a large increase in the number of questions clearly would help increase the statistical reliability of SEI evaluations, which currently can show major fluctuations based on very small numbers of "misses" of questions. But how well does it deal with the overall problem of ensuring that organizations will focus on concepts and issues, rather than building their processes around the questions themselves?

To answer such questions, let's look now at the issue how well *any* set of predefined yes/no questions (that is, an answer template) can be expected to perform as a method for evaluating complex systems. By "complex systems" we mean systems that have many parts that can interact with each other in a very large number of ways — a definition that clearly fits software processes!

Figure 7 provides a graphical example of the general problem of using fixed templates to evaluate complex, highly variable system. As can be seen from the figure, the central problem with such templates is that they can only "pin down" a small set of highly specific features. Even when there is general agreement that a specific feature is necessary (e.g., in Figure 7 the template checks for tires since it is hard to imagine a "good van" that lacks them), single-point checks are not easily able to constrain unforeseen (and unwanted!) variations of such features. One would generally not wish to define a device with forklift tires as a "good van," but if the data collected during the analysis is too sparse, it will become very difficult to prevent such excursions.

One might point out that there is a quite obvious way out of this — just increase the number of test points! It would only take more probe points in Figure 7 to banish the forklift dilemma. By adding enough new questions, one could eliminate essentially all unwanted deviations. However, this "more fixed data points" approach suffers from two significant problems.

The first problem is that adding a large numbers of new but still "fixed" data points will not only eliminate most non-vans — it will also eliminate most good vans! The reason is that fixed-point data collection requires not just a template, but also an ideal example of what a good van looks like. For example, if a car company decided to produce a new, people-oriented van that replaces traditional cargo-oriented sliding doors with conventional passenger doors, how would this new van rate under a fixed-template test? The answer is that it very probably would fail — not because it is a "bad" van, but because it is different from the particular van assumed to be "ideal."

It is worth noting that in the SEI testing scheme, the type of problem just described is aggravated because their original template is based on an "ideal" organization that *never existed.* The highest levels (4 and 5) of the SEI maturity framework are extrapolations of a very small number of individual projects done many years ago,[2] and the SEI literature gives little quantitative data on the "bottom line" productivity and quality results of those projects. Using such extrapolated information to build an "ideal" template is roughly equivalent to building one or two small models of a presumably ideal van, and then using measurements from those models to specify how the entire automobile industry should from now on go about producing "good" vans.

The second problem with simply increasing the number of questions is that it does not deal directly with the problem of using *fixed* data points to analyze *adaptive* systems — that is, systems which capable of changing their behavior in response to the testing activity.

For example, let's assume for a moment that the Figure 7 template is looking only for a "good" paint job, rather than an overall definition for a van. Since very little actual data is collected about the paint job on a van, it will not take the maker long to realize that doing shoddy work *or no work at all* on surfaces far away from the inspection points produces a van just as acceptable as one that was meticulously painted.

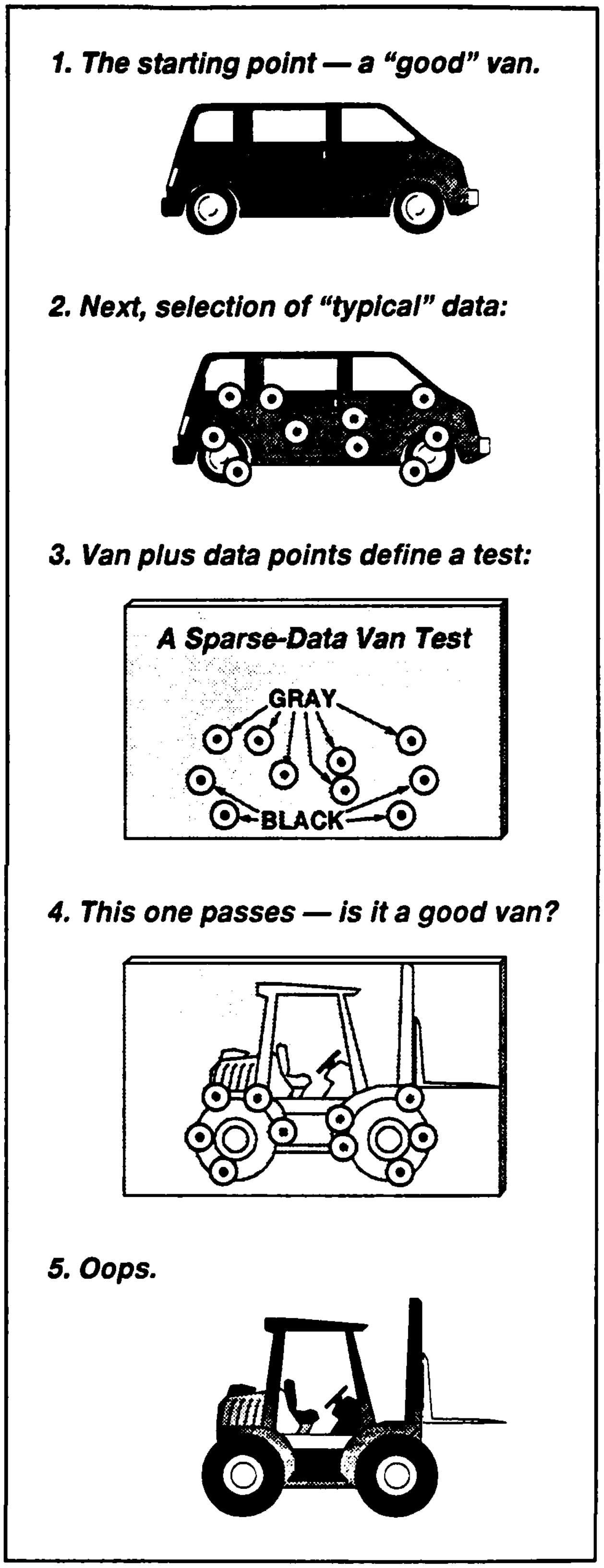

1. The starting point — a "good" van.
2. Next, selection of "typical" data:
3. Van plus data points define a test:
A Sparse-Data Van Test
GRAY
BLACK
4. This one passes — is it a good van?
5. Oops.

Figure 7 - Ambiguity in Sparse-Data Analysis

The absurd (but logical) end result of this thinking is for the maker to buy very precise, very high-quality "spot painters" to paint vans *only* at the inspection points. By investing heavily in high quality where high quality is "expected," the maker can build a process that gets very good quality ratings, and saves a bundle on paint costs. Meanwhile, the buyers of the vans wind up with very high quality polka dots, but little else!

The current SEI questionnaire is clearly subject to this scenario. For example, the three configuration control questions (2, 11, 12) back in Figure 6 imply that a configuration control function must exist for each project and must at least be able both to control changes to both requirements and code — but the questions say nothing about tracking of other intermediate products such as high-level designs, detailed designs, documents, test procedures, and sets of reusable test data.

The intent of these three questions appears to be that the configuration control system should track changes to a broad range of intermediate products, including such things as changes to requirements, designs, detailed designs, code, and test procedures. But because they are only a partial sampling, they actually specify tracking of changes only for requirements and code. As a result, an organization that is strongly motivated to move to the next higher level of the SEI model is quite likely to focus first on implementing these two *aspects* of configuration control, rather than dealing with the general concept. Indeed, because they are unfamiliar with the background of how the SEI questionnaire was set up, they may quite honestly assume that they are supposed to set up only these two aspects of a configuration control system. It is a difficult thing at times to question the knowledge of the tester, particularly when that knowledge is buried rather deeply in the structure of the grading system rather than the test itself.

Well-intended adaptation can also lead to some rather amusing grading effects, especially if they involve a question that is ambiguous, technically out of date, or just plain irrelevant. One example is given in Figure 8, which is based on an actual question from the 1990 SEI questionnaire. It shows how a sincere focus on a badly designed question can lead to responses that have little to do with tackling real productivity problems, but have a good deal to do with getting a better grade.

Behavioral scientists are familiar with the wide range of problems that arise when overly simplistic tests are applied to people or organizations, and they have a variety of sophisticated testing techniques to make sure that the results they get are meaningful. Needless to say, repeatedly giving exactly the same test with exactly the same answers to organizations whose financial life may depend on how they respond is not one of those techniques. Factors of this type need to be considered seriously in any attempt to build a new questionnaire for SEI evaluations.

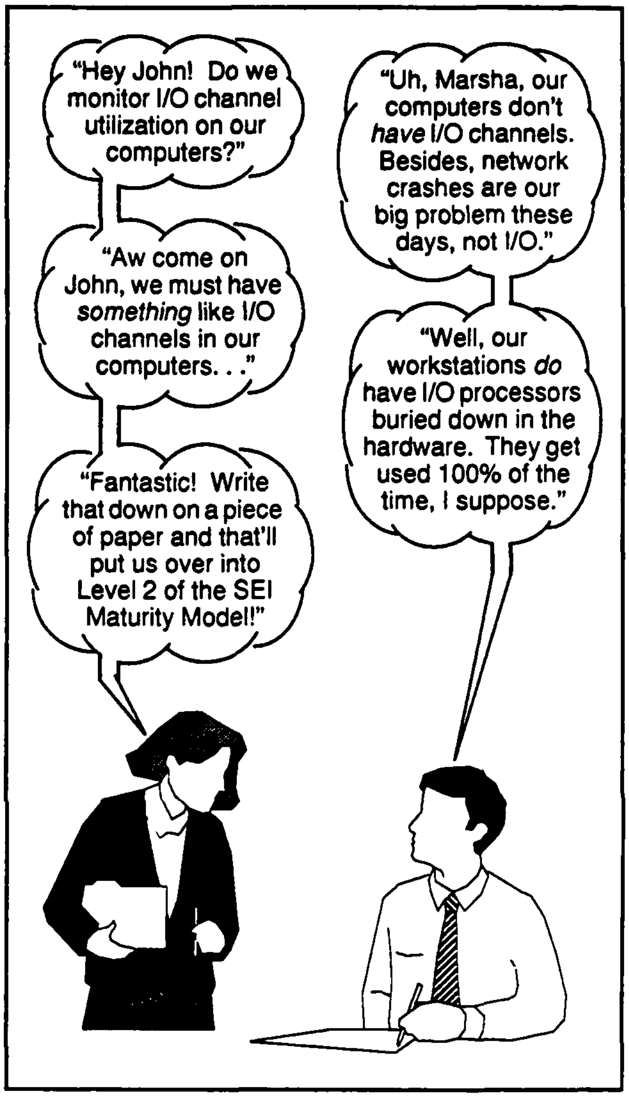

"Hey John! Do we monitor I/O channel utilization on our computers?"
"Uh, Marsha, our computers don't *have* I/O channels. Besides, network crashes are our big problem these days, not I/O."
"Aw come on John, we must have *something* like I/O channels in our computers. . ."
"Well, our workstations *do* have I/O processors buried down in the hardware. They get used 100% of the time, I suppose."
"Fantastic! Write that down on a piece of paper and that'll put us over into Level 2 of the SEI Maturity Model!"

Figure 8 - A "Worst-Case" SEI Grading Scenario

The above points by no means exhaust the potential problems that can derive from using a fixed template of questions to analyze a complex systems. As shown by the Figure 8 example, devising questions that are both highly specific and at the same time "universal" in their overall coverage of a topic is no trivial task, particularly when updates to the questions are few and far between. Specifically, large sets of yes/no questions

- are hard to generalize for use with a wide audience,
- are subject to rapid technical obsolescence,
- tend to encourage "fixes" of very limited scope,

- make it easier to overlook complex problems,
- can be highly unfair to groups with novel structures,
- can be ambiguous due to oversimplification of issues,
- can slow progress by forcing a myopic view, and
- tend to use jargon that can make them cryptic.

Sparse-Data Systems Analysis

A curious aspect of trying to use *any* kind of template-based approach to analyzing complex systems is that whatever approach is taken, the testing process seems to remain balky or unpredictable in some way. If the number of data points is kept low, the danger of wrongly identifying "bad" systems as "good" tends to become high. But on the other hand, if the number of data points is greatly increased, the danger of wrongly identifying "good" systems as "bad" tends to grow rapidly! What is it about such template-based approaches that seems to make them persistently misbehave when applied to the analysis of complex systems?

Much of the answer can be found by looking at such methods in terms of information flows during the analysis process. Figure 9 shows two quite different approaches to analyzing a complex system: a "standard" approach that is typical of most systems analysis work, and the SEI "sparse data" (or data template) approach.

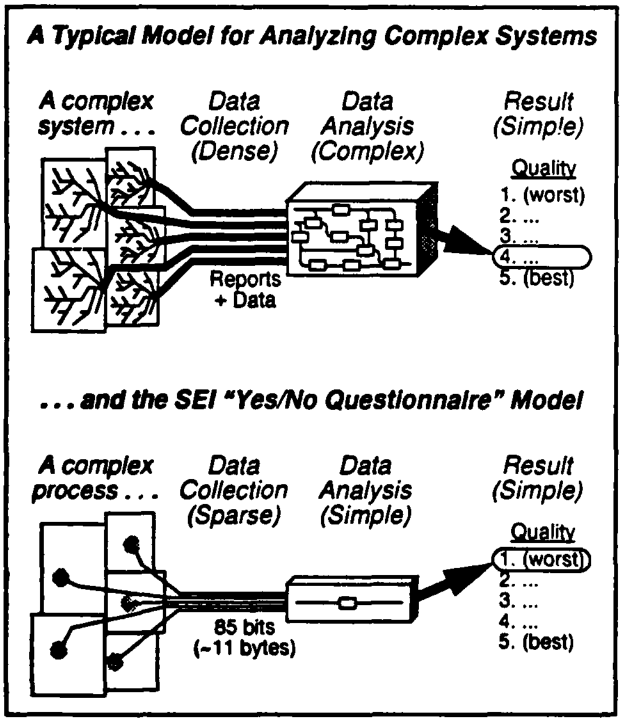


Figure 9 - The SEI Maturity Analysis Model

In the "standard" method, the data collection activity is both rich in detail and dynamic in nature. Even if the analysis begins with a general model of that the system will look like, the actual collection of data will depend to a large degree on the data itself. That is, data which shows indications of problems or important issues may be pursued down to a deeper level, or perhaps taken as a clue that further analysis is needed in some other area that may at first have seemed unrelated. The data collector must play the part of a "data detective" — she or he must follow leads wherever they go, search for hidden clues, and generally try to expect the unexpected. If the case is simple, it may be possible to wrap it up fairly quickly — but if it is a system involving people, the data collector is likely to uncover a good deal of work to do.

Because it usually involves unexpected twists and turns, the data collected by such a process is itself rich and complex, and must be represented in formats that are able to record that complexity adequately. Reports, diagrams, and various tables of data thus may be used.

The analysis step is also complex, because it has the task of converting this rich set of data about a complex system into some set of relatively discrete "bottom line" recommendations. The data does not disappear, but it must be categorized and grouped in ways that show what is important and what is not. In many cases, it will not be the "whole picture" data that decides the case — in fact, it is perhaps more likely than not that it will turn out to be some of the smaller, less obvious information trails that turn out to be critical to the final decision. Anyone who dealt with an unscrupulous used car dealer knows this, although they may not discover the key clue or question until it is too late!

Ironically, a good example of this type data-rich analysis process is the SEI Process Assessment program. In SEI assessments, the emphasis on non-attribution and the skill of the SEI process group provides an atmosphere that tends to be rich in data and insights about the software process. This information is then captured in the form of a report that is also relatively rich in information, and which gives the vital bottom line recommendations needed to convert ideas into actions.

The irony is in how differently the system analysis process works for the parallel SEI evaluation program. One difference simply that of the consultant-versus-auditor analogy. Tax auditors tend not to have much success at cultivating an atmosphere of open information flow, however good their intentions may be. But more importantly, SEI evaluations differ from assessments because they *must* convert all of the results of interviews and discussions into 85 one-bit, yes/no questions. This means that the final analysis process is based not on a rich set of reports and complex tracings of problems, but on roughly *11 bytes* of binary data, as shown in the lower part of Figure 9. (SEI assessments also fill out this form, but are under no obligation to make it the "only" result.)

An obvious response to the Figure 9 depiction of SEI evaluations is that it ignores the intensive interviewing and analysis process that occurs *before* the data is finally "wrapped up" a mere 11 bytes. Isn't this simply a case where data analysis has been moved up to a slightly earlier stage, with the SEI grading system performing just a minor "reformatting" of the 85 bits into SEI levels?

Unfortunately, no. Raw data in systems analysis is best defined as data that arrives in independent pieces or "chunks" that have not yet been strongly correlated or compared. Individual chunks may be quite complex, but they are still raw data in that they have not been weighed and correlated against each other to identify underlying causes and issues.

As an example of what this means, take the case of a software maintenance process that is being analyzed in hopes of improving its efficiency. One chunk of raw data for such a system might be a "traceback" of why test schedules are persistently underestimated for one rather specific type of change requests, but tend to be quite accurate for other types. If this input to the analysis phase consisted of little more than the assertion that 10% of all test schedules were underestimated, it would be very difficult for the analysis activity to determine just what the reason for this might be. Is it a process problem that can only be corrected by replacing the current estimation procedures with new ones, or is it instead related to some aspect of the software product? If the data chunk is too sparse, it becomes difficult to correlate it with other results about the process and the product.

On the other hand, if the data chunk is rich in its level of detail about the estimation problem, it will include a good deal of information about the *type* of change requests that tend to cause estimation problem. By correlating that information to data chunks about known problems in the software that is being maintained, the analysis team will be able to identify that the underestimated tasks all involve changes to one set very poorly coded modules. The resulting process level recommendations in such a case would be that the estimation process be augmented to recognize and estimate these cases more accurately, and that an active program of proactive redevelopment of trouble-prone modules might be needed.

In the case of the SEI analysis model, the types of issues covered by the yes/no questions are much closer to being uncorrelated, raw-data chunks than they are to final conclusions, since they deal with 85 individual samples of broad issues whose exact relationship is very much open to debate. (The SEI model attempts to impose order on these samples by using the assembly-line paradigm of manufacturing cars, watches, and sheets of metal, but as we shall see in the last section of this paper, there are some very serious problems with trying to use the assembly line metaphor as the basis for analyzing software process data.) In terms of sparseness, these 85 data chunks represent the ultimate extreme — they each consist of only one bit!

Once again, it becomes an issue of statistical reliability. If the 85 bit filter happens to "match" the crucial points of a process, it may give good results. But if the critical pieces of information about what is wrong (or right) with a process fall outside of 85 bit filter, the results will be valueless. The earlier example of how one group might automate math-to-code translation while another group does it by hand provides a simple example of the impact of such data losses. Because the lower levels of the 1990 SEI template are "blind" to process automation issues, it could give the human-intensive group a higher rating simply because it produces more data of the type expected by the SEI template.

Once the 85 bits of raw data have been collected, the actual analysis becomes an extremely simple process by systems analysis standards. No significant correlation or weighing activity is needed or appropriate, since each chunk of raw data is only one bit. Thus a very simple, algorithmic "analysis" activity becomes possible. For the 1990 SEI questionnaire, this analysis takes the form of the seven-hurdle, five-level grading system previously described in Figures 3 and 5. Ironically, the SEI system is actually *too* complex to reliably process its severely stripped-down 85 bits of data, as described in the earlier discussions on the statistical unreliable method of using chains of small grading hurdles that "chop up" the test.

The first part of this paper has been devoted to the issues of SEI assessments and evaluations, and especially to the testing model (maturity levels, questionnaires, and grading systems) that define the core of SEI evaluations. Before going on to the broader issue of the overall direction and process maturity goals of the SEI program, Figures 10 and 11 provide a brief summary of many of the points that have been brought up, phrased in terms of the striking contrast between the outstanding work SEI has done in building their process assessment program, and the curious inversion that occurs when that same general method is made into a sparse-data grading system.

Dr. Jekyll: SEI *Assessments* (1990)

- **Rigorously developed.** The SEI process team has developed an excellent program of extensive interaction and feedback. Problems in training or the assessments themselves are collected and fed back to help improve the SEI training program.

- **Well-received.** Although most organizations and people are initially reluctant to undergo process assessments, SEI has done an outstanding job of setting up the interview process so to encourage communications without making people feel they are being singled out.

- **Software engineering technology transfer.** Another important effect of SEI assessments is that they tend to make those involved much more aware of and interested in software engineering concepts and techniques.

- **Promotes "process awareness."** SEI process assessments also promote a general awareness that many types of problems are actually *process* problems, as opposed to *people* problems. This can lead to substantial short-term improvements by allowing different groups to stop "blaming each other" and instead focus on changing the process.

- **Better communications.** One of the most notable results of SEI assessment is that they nearly always result in better communication between within organizations and projects

- **Forward-looking perspective encouraged.** By allowing people to freely air old grievances and simultaneously work towards solving them, the SEI evaluations tend to promote a positive, forward-looking perspective on what they can accomplish in the future.

- **Team outlook promoted.** In SEI assessments, solutions are arrived at jointly by members of projects, functional groups, and management. This joint effort usually carries over after the assessment as a feeling of increase team spirit.

- **Long-term improvement.** By collecting results into a management-approved report, the long-term prospects for improvement are improved. (Note: SEI reports do not record the process.)

Figure 10 - Major Features of SEI Assessments

Mr. Hyde: SEI *Evaluations* (1990)

- **Results based *only* on SEI questionnaire.** The data used determine the process maturity comes *only* from the 85 graded yes/no questions of the SEI questionnaire. Favorable results in a process assessment do *not* ensure good test results.

- **Penalties for low grades are severe.** When the SEI system is fully in place, a company that does poorly on the maturity test could be barred from competing for DoD contracts, *even if their record of actual software delivery and quality is excellent.*

- **Built on an *unproven* maturity model.** Rather remarkably, the five-level SEI maturity model was *not* derived from some highly effective "archetype" Level 5 organization (there was and is no such organization); nor does it represent any kind of general consensus about good processes by the software community; nor did SEI made any strong effort to verify the model experimentally before making it "standard." The higher levels (4 and 5) were derived entirely from anecdotal examples of a very small number of projects, and the SEI book that describes levels 4 and 5 provide very little in the way of performance data for those projects.

- **Grading is based on very sparse data.** Since the SEI test uses only 85 *yes/no* questions, all the complexity of a large software organization must be "squeezed down" into 85 bits (about 11 bytes) of information before it is graded.

- **Random questions.** Because there are so few questions, the SEI approach was to sample a variety of features in organizations. The result is that organizations tend to respond to SEI grading by implementing quirky process improvement programs centered on the SEI *questions*, rather than on the underlying concepts.

- **Quirky grading system.** Because it is based on such a small number of questions, the seven-hurdle SEI grading system can easily result in unfair maturity classifications. As an extreme example, an unprepared organization could answer yes to all but two of the 85 questions *and still be classified as an "unrated" Level 1.* More ominous is the fact that organizations that home in on key questions probably could achieve unrealistically high SEI ratings.

Figure 11 - Major Features of SEI Evaluations

The SEI Process Improvement Model

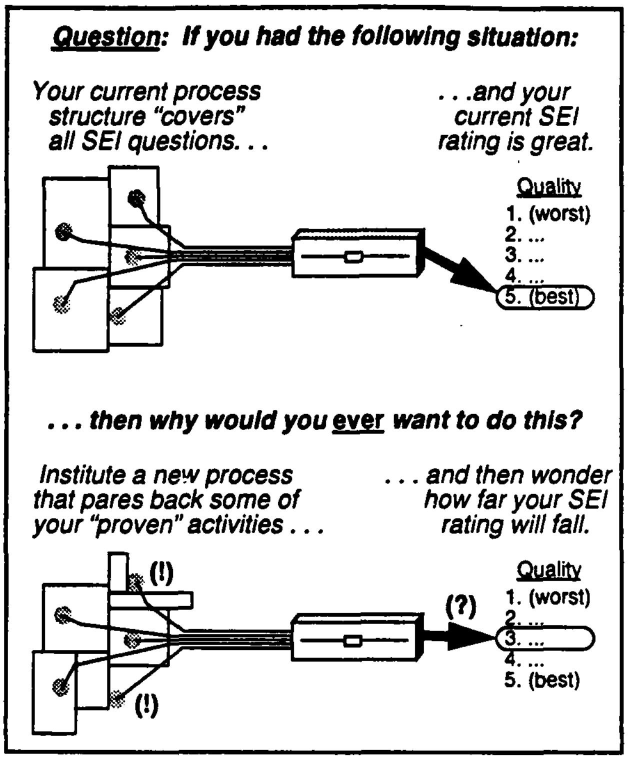
The first part of this paper has been devoted to SEI assessments and evaluations, with a particular focus on the grading system used in evaluations. It is now time to take a closer look at SEI's overall *process improvement paradigm* — that is, the set of goals and directions that it is trying to impart to the industry through its assessment and evaluation programs. The question is this: Where, exactly, is the SEI process improvement paradigm likely to take the software industry if it is fully (and presumably correctly) implemented?

A first point that needs to be made is that SEI's process improvement goals cannot be fully separated from SEI's testing program, since their testing methods have such a strong influence on how organizations will view process improvement. For example, the implications of sparse-data systems analysis also extend to process improvement. Figure 12 shows graphically how the use of a sparse-data template can significantly influence the overall direction of process improvement for the industry.

The point of Figure 12 is when a process works well with the SEI testing template, its manager is going to be reluctant to change it very much. The issue becomes one of benefits versus risks. If the manager removes a process activity that is wasteful, but which happens to produce data that matches the 85 bit template, she might benefit by a productivity increase of a few percent. Unfortunately, she will also be aware that the wasteful process is very "mature," and thus adds significantly to the overall chances that the next SEI evaluation will give a favorable result. The risk then becomes that removing the process could result in a lower SEI score — a result that could cost her the entire contract, and possibly future contracts as well. Given such a situation, most managers will opt for the conservative approach of no change — particularly if the contracts happen to be of the cost-plus-fixed-fee variety.

This effect of *process fossilization* is a very real risk for any evaluation method that relies primarily on fixed templates for its data collection activity. (Data-rich analysis models avoid this problem by being far more flexible and intensive in their approach to acquiring data.) The irony of finding fossilization effects in SEI's process improvement model is that its central objective is to *optimize* processes, not fossilize them.

As it turns out, though, the sparse-data grading system is not the only part of the SEI evaluation package that encourages process fossilization. Another significant fossilization effect turns up in a most unexpected place: Level 5, whose defining feature is that it is supposed to help "optimize" software processes!



Question: If you had the following situation:

Your current process structure "covers" all SEI questions. . .

. . .and your current SEI rating is great.

. . . then why would you ever want to do this?

Institute a new process that pares back some of your "proven" activities . . .

. . . and then wonder how far your SEI rating will fall.

Figure 12 - The Process Fossilization Effect

To understand how this could come about, we must first take a closer look at just how Level 5 is defined. The questionnaire definition of Level 5 is not very helpful in this case, since it "captures" Level 5 in only four rather generic questions. However, Chapter 17 of *Managing the Software Process*[2] provides a detailed description of Level 5, and also provides helpful examples of several Level 5 process optimization forms. These forms are to be used by projects to help translate the occurrence of errors in software *products* into specific actions to help increase the reliability of the software *process.*

Based on these descriptions and examples of forms, the optimization model of SEI Level 5 is shown in Figure 13. Whenever a specific defect is identified in a software product, the exact nature of that defect is identified and recorded and a "traceback" (the authors' term, not SEI's) is initiated. The purpose of a traceback is to identify as specifically as possible where the defect originated, how it occurred, the underlying (process) cause for the error, and how the process should be changed to prevent that particular type of defect from occurring in the future.

The authors have introduced the term "traceback" to emphasize that the fundamental concept of Level 5 is neither terribly complex nor hard to understand. Indeed, it is something with which anyone who has debugged a program can claim considerable familiarity. Because in the SEI model it is applied to *processes* rather than *programs* does not change its fundamental features. Nor does it change some of its fundamental limitations!

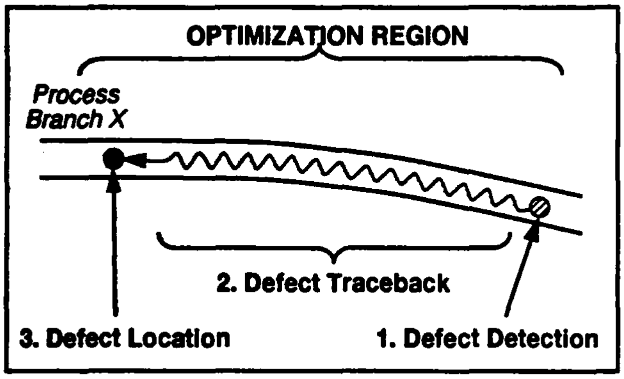


Figure 13 - SEI Level 5 Optimization Paradigm

The problem with tracebacks is that regardless of whether they are applied to programs or to processes, they remain essentially a *local*, or "bottom up," approach to optimization. Tracebacks are "local" in the sense that although they may cross boundaries between activities, they tend to provide little or no information about the overall quality or reliability of a program or process. For example, it is possible that the Branch X in the diagram represent an entire stream of activities that are actually redundant and should have been eliminated.

Even without further analysis, the observation that SEI's Level 5 is just traceback optimization should be a cause for concern. Although it is a vital technique for debugging programs (and presumably processes), traceback by itself is a truly terrible way to *restructure* a program (or process). More often than not, traceback optimization in programming results in a gradual degradation of quality, not an enhancement. This is mostly because its myopic focus on fixing isolated defects or bugs leads it to "violate" all sorts of tacit agreements in how the program should be structured. Formerly pristine interfaces may be violated, and key information hiding restrictions may be damaged in subtle, hard-to-track ways. In the current state of the art, software processes are generally not clearly defined as programs, but they certainly include their own types of "agreements" and information sharing arrangements between activities. As with programs, a process optimization strategy based only on traceback is likely to cause a gradual degradation of the rules and agreements — many of them implicit — that help a process to perform predictably.

In the case of the SEI five-level maturity model, it can also lead to a second type of process fossilization. The SEI model requires that processes that wish to reach Level 5 must first undergo an intensive "instrumentation program," in which a wide variety of data is collected throughout the software process. This trend is seen as early as the very first hurdle in the SEI grading system, in which two of the twelve questions are devoted to data collection issues. By the time Level 5 is reached, each branch of the process has been "instrumented to the hilt" with a variety of metrics for measuring product quality and process efficiency.

The result is a marvelous case of self-deception. Since each process branch is so extensively instrumented, defects are indeed found and methodically corrected to ensure "clean" passages. The process is similar to a massively debugged program that has been instrumented with all sorts of debug statements that constantly churn out data that "prove" the program is working correctly. The process (or program) is working great, and its extensive network of instrumentation is ready to catch any new defects that might pop up.

What all these handy measurements do *not* reveal is that the overall structure of the process (or program) may be in an absolutely horrible mess, ready to collapse the first time anyone attempts to make any serious change to it. When viewed from the outside, the various branches of the heavily instrumented process (or program) may lack a clear structure, since any explicit or implicit partitioning agreements made early in the history of the process (or program) are likely to have been violated. The process (or program) has been optimized, all right — but only at the expense of throwing away its ability to do anything else except *exactly* the role into which it has evolved. And it is inflexible even in that, since global optimizations would involve major structural changes that could result in a similar type of collapse.

This then is the fossilization effect of SEI Level 5 — the very real danger that a process that uses a combination of "bottom up" traceback optimization with extensive instrumentation will also wind up with a highly inflexible, "bottom up" design for their software process. This is effect is further enhanced by the fact that the bottom-up process structure has also become an excellent "data factory" for meeting the requirements of a given sparse-data test template.

How can such a scenario possibly be reconciled with the very explicit statements in the SEI literature that Level 5 is supposed to be both an innovator in process and technology?

Actually, quite easily. It might be termed "the SAGE effect," in honor of the highly effective (but dated) vacuum tube computer system that was a key part of our national defense system until the mid 1980s. In a typical Level 5 process, each new technology will be rapidly and efficiently evaluated — but the extensive data available through process instrumentation will then be used to prove conclusively that the technology is inappropriate for that process! Only technologies that result in very minor tweaks to the "nearly perfect" process will be able to squeak through such a daunting data gauntlet.

Another interesting aspect of the SEI Level 5 traceback paradigm is that *there is no obvious reason why it could not be used effectively at the earliest stages of process improvement.* The nominal justification for placing it in SEI Level 5 appears to be that an abundant supply of data is needed to make it work. However, traceback is essentially an inductive, "detective" technique that relies more on insight and the following of clues than it does on does on having masses of data available to it. Indeed, there are more than a few cases in which large masses of data may tend to obscure, rather than help, the tracing activity.

The authors have found by experience that traceback is a very helpful technique during the interview phase of a process assessment. Since assessments provide access to a wide range of people who normally do not talk to each other about their work, it is very common to encounter some set of consistent themes or problems that can eventually be traced back through the entire process until a specific process cause can be identified. Clearly this is not a rigorous, numbers-first approach to using traceback — but it is without a doubt a very good way to identify pivotal problems in a software process!

Indeed, one could make the argument that "up front" is the *best* location for applying traceback optimization, since it is a excellent technique for identifying the types of major mismatches or missing activities that are most likely to occur early in the history of a software process. The problem with using traceback only in the later stages of a software process is that by that time all of the early indicators of major process flaws will have been covered over by a series of lower-level fixes. By the time traceback optimization is applied in the SEI model, it will no longer have sufficient power to identify much more than minor, locally generated defects.

A Framework for Global Process Optimization

There is still one very important aspect of the SEI process maturity model that needs to be discussed, and that is its fundamental premise. The premise is that the quality and process control mechanisms of assembly-line manufacturing can be applied more-or-less directly to software processes, and that the result will surely be the same type of phenomenal quality and productivity improvements that have been seen in recent decades in the manufacture of watches, cars, sheet metal, and other mass-produced, assembly-line products.

This premise happens to be wrong, but it will require a bit of background preparation to explain exactly *why* it is wrong. The earlier discussion of why SEI Level 5 is more likely to lead to process fossilization than to process optimization provides a number of valuable clues that something is very seriously wrong, but a full explanation and quantification will have to wait till later in this paper.

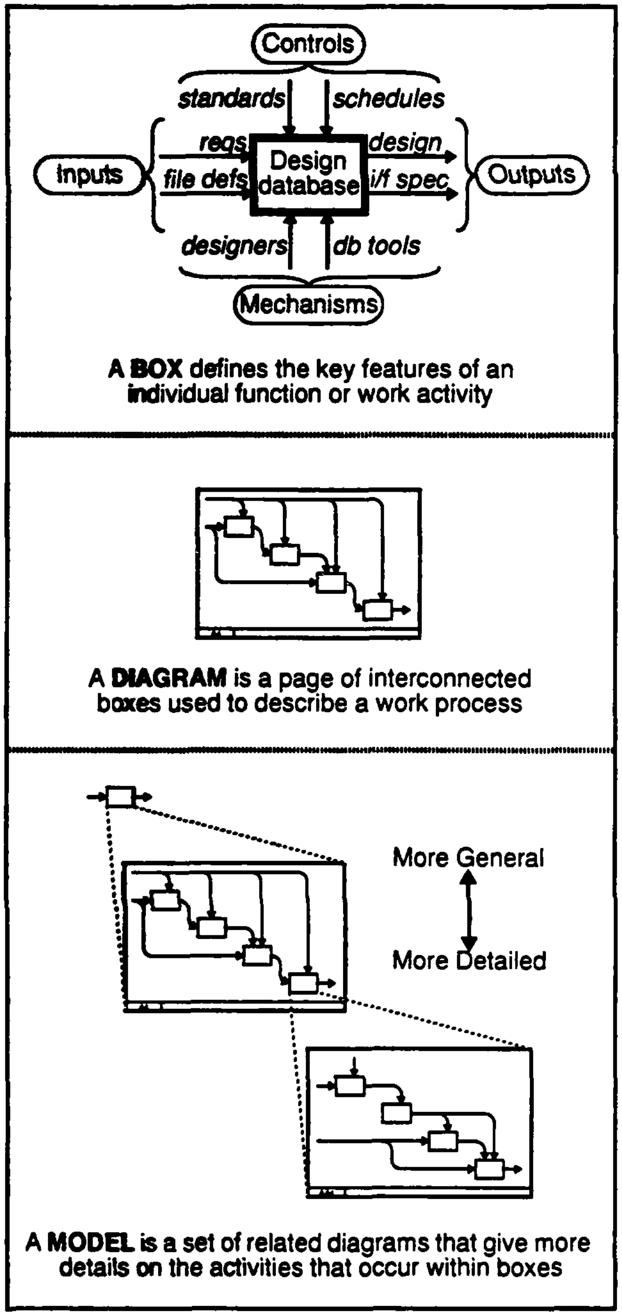
To provide that necessary framework, let us now turn our attention for a while to the very interesting and important problem of *global* process optimization. This section of the paper will introduce a number of concepts and terms that have been derived from a combination of actual experience in performing modified (non-SEI) process assessments on several diverse software projects. This experience base was combined with a subsequent analysis of those results to look for common themes and issue. The material is thus theoretical — but it is theory that is based on firsthand experience in analyzing "real" projects.

Mapping Processes Structures with SADT

In discussing Level 5 of the SEI model, we pointed out that there are a number of close parallels between the optimization of *process* structures and the optimization of *program* structures. However, one clear difference between the two is that programs can be written out and analyzed on a screen or a printout, while processes are usually nebulous entities that are far less easy to see and grasp. The first tool that is needed to discuss global process optimization, then, is some way to cleanly and unambiguously *write down* the overall process structure. Once it can be seen on a screen or a piece of paper, the whole idea of performing global changes to a process becomes much more concrete and understandable.

There are probably many ways this could be done, but an approach devised by one of the authors (McGowan) has proven particularly convenient for recording process data in a format that is easy to use and readily understood by members of the project or organization that is being analyzed. The latter property is very valuable, since it makes it much easier to verify the accuracy of a process description and explain recommendations.

The method is the Structured Analysis and Design Technique, or SADT.[4] A very brief explanation of SADT conventions may be found in Figure 14. As applied to software process, SADT consists of boxes that describe individual activities in a process, and labeled arrows that represent various type of inputs (such as products, resources, or controls) or outputs (transformed products) that flow through the activity boxes. By definition, three sides of each box are always used for specific types of inputs (products, controls, and resources), and the remaining side is always used for the (transformed) outputs of the activity. The only other major addition to the rules for SADT diagrams is that boxes can be expanded into separate SADT diagrams, so that an activity can be described down to whatever level of detail is most appropriate for that process.



A **BOX** defines the key features of an individual function or work activity

A **DIAGRAM** is a page of interconnected boxes used to describe a work process

A **MODEL** is a set of related diagrams that give more details on the activities that occur within boxes

Figure 14 - SADT: A Way to Picture Processes

Figure 15 shows an actual SADT process diagram developed during a process analysis that was performed for a software maintenance project. Although such diagrams look complex, they are generally very readily understood by participants in the process — after all, they are essentially just "road maps" of the activities that normally go on within the process. The particular diagram shown is an example of a *context diagram,* which is always the first diagram in a process model. Its purpose is to show the overall environment in which a software process is imbedded. Such context diagrams permit a better understanding of the many external constraints that may drive or modify a software activity.

In many cases, opportunities for global optimization of a process can be identified simply by inspection of an SADT process model. For example, one common process problem that often may be spotted by inspection is the presence of redundant activities. Two or more boxes in the model may be doing very similar work, but may not be aware of it due to poor communications or different terminology. Besides being inefficient, such redundant work activities can lower average product quality by creating more than one product with the same purpose. As a rule of thumb, highly redundant work activities should exist in a software process only when there are specific customer requirements (e.g., the need for very high levels of software reliability) that make them necessary — and even then they should be very tightly defined and monitored.

Top-Down Process Optimization

Since SADT process models are hierarchical, they easily lend themselves to a *top-down* approach to process optimization. This simply means that when looking for ways to simplify or increase the reliability of a process, the highest level (context) diagram is always examined first. Because processes may in some cases consist of multiple boxes even at their highest level, starting with the context diagram is particularly important because it always shows such processes as embedded in closed systems. This kind of closure at the highest level leads naturally to analyzing the global effects of change, not just the local effects.

After a reasonable set of optimizations have been selected at that highest level, the optimization activity then proceeds down to the next level of diagrams to look for the next set of optimizations. If necessary, this top-down optimization strategy may continue all the way to the lowest level activities of the process. In other cases, it may decided to end it at a higher level if because the benefits of lower-level optimization would be minimal.

Regardless of whether it is based on an SADT model of the process or some other technique, the importance of using a top-down strategy to optimize a process cannot be overstated, since problems that appear at the lower levels of a process are often just consequences of higher level problems. A common example of this effect is when an inadequate "up front" definition of customer needs results in chaotic design and coding activities later in the process. No matter how orderly the design and implementation activities may seem to be in such a case, chaotic behavior in the overall software process will persist for a very simple reason — the designers and implementers are building the wrong software!

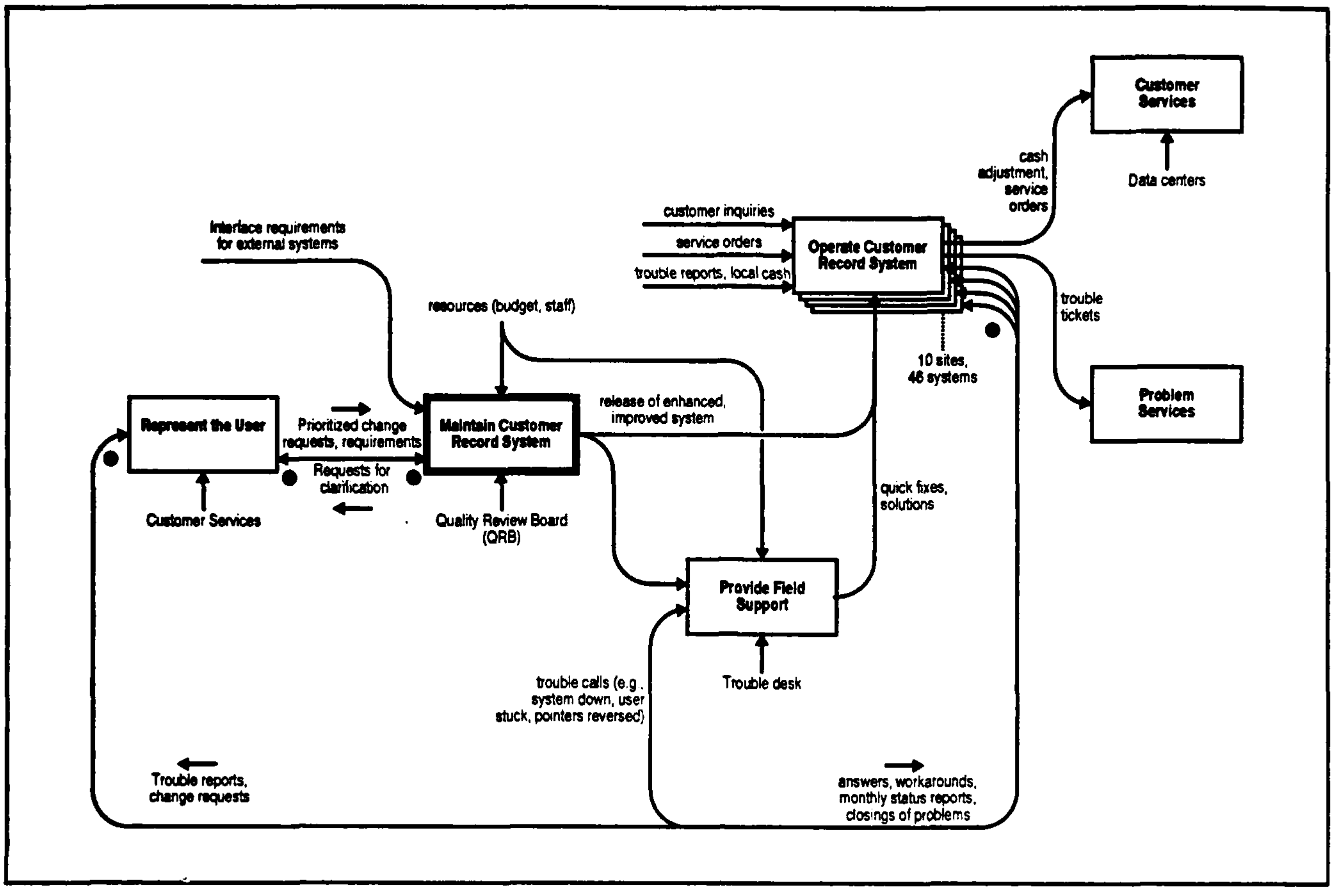

Customer Services
Data centers
cash adjustment, service orders
customer inquiries
service orders
trouble reports, local cash
Operate Customer Record System
Interface requirements for external systems
resources (budget, staff)
trouble tickets
10 sites, 46 systems
Problem Services
Represent the User
Prioritized change requests, requirements
Maintain Customer Record System
release of enhanced, improved system
Requests for clarification
Customer Services
Quality Review Board (QRB)
quick fixes, solutions
Provide Field Support
Trouble desk
trouble calls (e.g., system down, user stuck, pointers reversed)
Trouble reports, change requests
answers, workarounds, monthly status reports, closings of problems

Figure 15 - An SADT Process Definition Diagram

Process dithering

If process optimization is not done top-down, it is likely to result in *process dithering.* Process dithering is when a project or organization spends most of their time trying to optimize or improve the low-level symptoms of what is actually a high-level flaw in the process. A simple example would be a project that spends a great deal of time and money to instrument and measure a particular branch of their process, only to discover later that the branch is redundant and should have been cut out altogether.

Process dithering is a significant temptation in any kind of process analysis, since it can provoke furious activity and produce reams of "process improvement data" — all without having much real impact on either the efficiency of the process or the quality of the software it produces. Top-down optimization using SADT process models is a good way to avoid process dithering, since SADT models always map out high level relationships first.

The phenomenon of process dithering also help point out why *product quality* cannot readily be separated from the issue of *process efficiency.* A process that relies on process dithering to produce high quality products may in some cases work, but only at the cost of making the process fragile and inflexible. Since it does not truly fix major process problems, process dithering instead tends to result in a process that needlessly creates and then tears down problems that never should have existed in the first place. Changing such a structure then becomes risky, since any significant modifications may upset this peculiar system of checks and balances of needless problems.

In short, an efficient process tends to be simpler, cleaner, and more flexible towards change. And in the long haul, those are the kinds of features that are most likely to result in truly high-quality products — particularly for software, where flexibility and change are the name of the game.

API — A Method for Designing Process

With the background of SADT diagrams in hand, it is now time to develop a framework for analyzing the efficiency and quality of a process in a more quantitative fashion. We call the particular set of ideas you are about to see the Allocator-Producer-Integrator (API) method. API is still in its early stages, but it nonetheless provides a good kernel for the idea of designing software processes, rather than just reacting to *ad hoc* product requests.

In API, new software processes would be designed in a top-down, recursive fashion. Each level of the design would be structured not only to meet the specific needs of a customer or set of customers, but also to deal explicitly with varying levels of risk and product variation. The API framework includes assembly-line processes, but only as a specialized case of the overall process design model.

Figure 16 shows the basic concept of dividing processes into three major categories: *allocators, producers,* and *integrators.*

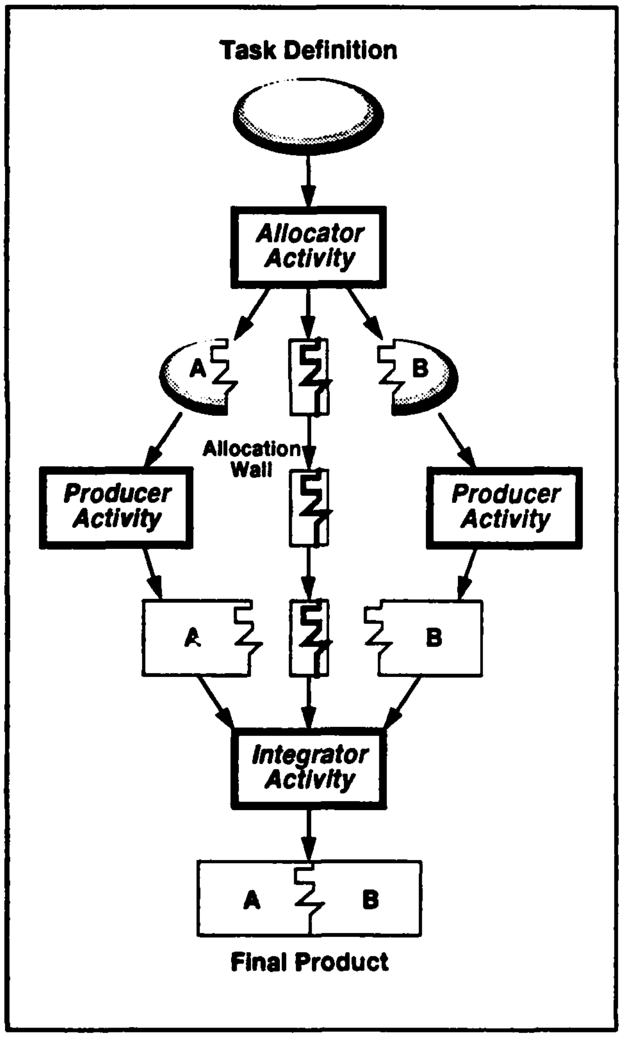


Figure 16 - Allocators, Producers, and Integrators

- **Allocators** are simply activities that are responsible for dividing up some task into two or more subtasks.

  The classic example of an allocator in the software process would be the high-level design of a software package. The design activity in that case would allocate various subsets of the original customer requirements as subtasks.

- **Producers** are responsible for receiving tasks (work assignments) and transforming those tasks into specific products.

- **Integrators** are responsible for re-assembling the results of producers into a product that meets the original task definition of the allocator.

In addition to defining subtasks for producer activities, the allocator activity is also responsible for creating the *allocation wall,* which is an abstract communication barrier between its producer activities. An API diagram may either show the allocation wall explicitly or leave it as an assumed component, but in either case its role in assessing the behavior of a given process is critical. A process with a "weak" or "leaky" process wall tends to be costly and unpredictable, and the resulting products tend to be of poor quality. On the other hand, a strong allocation wall tend to result in a much more predictable process and a superior quality product.

As shown in the diagram, an allocation wall is often (but not always) created by building a *parts interface.* Thus in a process that uses Ada, an allocation wall might be created by designing a set of package specifications and an accompanying set of functional descriptions.

One should not confuse *part interfaces* with *process allocation walls.* Part interfaces are a product level issue that deal only with how the parts will join together to product the final working product. *Allocation walls* are process level constructs whose purpose is to *prevent* two or more working groups from communicating with each other during development of a product. The success of a product interface definition is measured by how well it *supports* communication or interaction between parts of a *product.* The success of an allocation wall is measured by how well it *prevents* communication between parts of a *process.*

The confusion between the two arises in part from the fact that one of the best ways to build a good allocation wall is first to build a good parts interface, but our lack of familiarity with the process perspective is clearly an important factor also. When a designer goes down the hallway to ask someone in another group about how they are designing a part, we do not normally think of such an action as a "data flow" in the same sense as we think of program data viewpoints. And yet from a process viewpoint, such a conversation is very definitely a type of data flow. Important information is being passed between nominally separate activities, and the passing of that information may in some cases have a major impact on the cost of the process and product quality.

Thus saying that an allocation wall is strong means that at the time the product is being developed, little or no communication about its design occurs between its producer activities. In the same way, saying that an allocation wall is weak means that very extensive communication about the design of the product goes on between parallel producer activities.

Communication in this case is literally measured by how many *unplanned* exchanges of significant technical data took place. Such exchanges could take place via meetings, memos, phone calls, e-mails, or other media, but they all share the characteristic of being unanticipated by the allocator activity when it originally set up its subtasks.

Figure 17 shows two structures that are used to link allocators, producers, and integrators together. The first structure is called a *fan,* and it corresponds to the type of allocation into parallel activities that has already been described for Figure 16. The second structure is called a *pipe,* and it corresponds to the idea of an assembly-line style of development in which a product is passed progressively down a "chain" of development. As with a fan, the allocator activity in a pipe is responsible for preparing all necessary interface definitions before the task is passed down into the chain of producer activities.

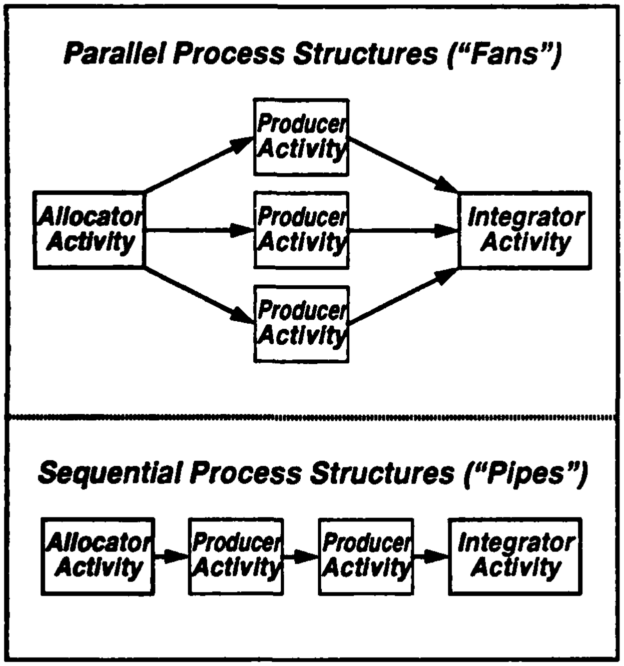


Figure 17 - Process Structures: Fans and Pipes

One might note that Figure 17 does not include any kind of a looping construct. In API both the fan and pipe examples of *monotonic* structures, meaning that they move the project smoothly towards completion. A loop in API is *non-monotonic*— that is, it represent some type of "back pedaling" or loss of work. Figure 18 shows three examples of such loops, with each loop labeled by the amount of *lost work* that results from such a loop being activated.

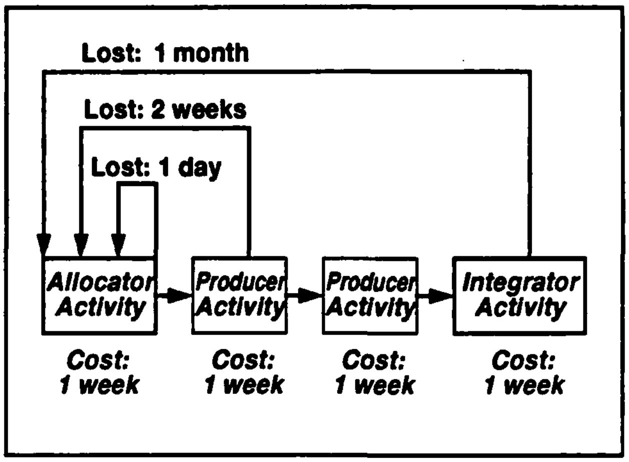


Figure 18 - Process Loop Reduction

Lost work turns out to be a very useful way to measure the effectiven:ss of a software process, since it can be applied at any level of the process and is not fooled by *locally* good results. After all, if a customer rejects a product that was generated by a smooth, orderly software process, the result is still a significant loss of work, no matter how clean the process was. Focusing on lost work in such a case would place the attention where it belonged — in the initial analysis of customer needs.

The overall point of Figure 18 is that in terms of lost work, there is a very strong process design incentive to shrink loops to as small a size as possible. Note that this action of *loop reduction* is not necessarily the same as *loop elimination.* Loops provide a very important function in API — specifically, they help reduce risk. Loop reduction seeks to find a balance between risk reduction (which pushes towards multiple loops) and process efficiency (which pushes towards monotonic, zero-loop process structures). Ideally, looping behavior should be kept entirely within the allocator activity, which is just another way of saying that the allocator should be able to "build" a strong allocation wall.

The need for loop reduction also helps explain why API places a strong emphasis on unplanned communications between producer processes. As shown in Figure 19, such interactions are the most common type of "trigger" for activating costly lost work loops.

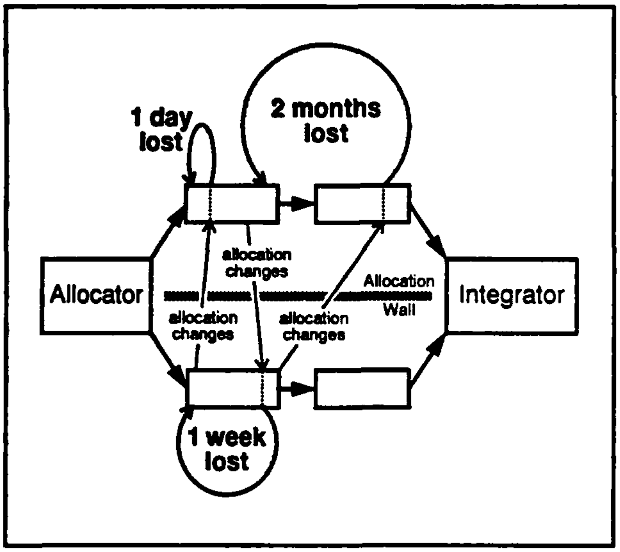


Figure 19 - The Cost of Unplanned Interactions

Although some classes of technical interchanges such as sharing programming techniques are rather harmless, any kind of a technical exchange that includes an inadvertent or intentional *allocation change* is likely to have a process impact. In particular, such exchanges usually introduce some level of looping in the API representation. Even worse, such exchanges can easily result in confusion about the "official" definition of the subtasks, resulting in even more looping behavior farther along in the process.

Some idea of the nature of the allocator activity is given in Figure 20, which points out that *stability over the duration of the process* is one of the key features that it must either design or uncover to ensure a smooth process with few interactions. Although such a criterion sounds simple, it can be remarkably difficult in practice.

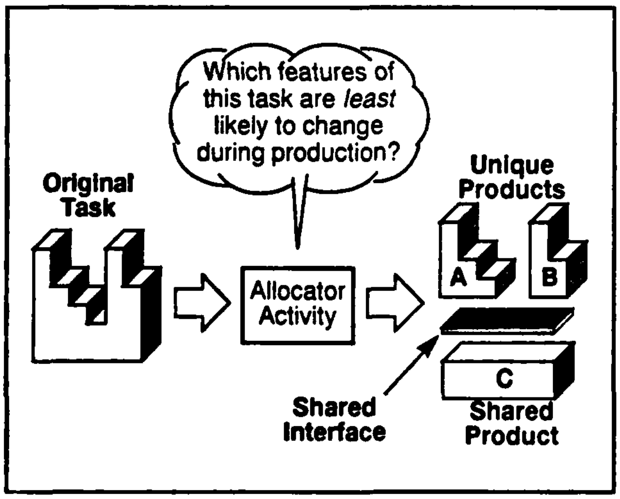


Figure 20 - Allocation as "Task Factoring"

Nonetheless, this concept of *task factoring* provides a useful guideline for understanding how well a design or interface specification is likely to "hold up" during the rest of the software process. In some cases, simply asking questions in terms of *how stable* a definition is likely to be can provide valuable guidance as to how the allocation wall can be strengthened. For example, prematurely setting an interface definition simply to "have something" is a very bad idea if the definition is to be part of an allocation wall — but it could be a good idea if it is part of a low-cost loop that occurs entirely within an allocator.

This completes our overview of both SADT as a process specification and optimization tool, and of API as a tool for analyzing processes in terms of issues such as risk and process efficiency. Although there is a great deal more to do to fully form these two methods into a general process design methodology, they already provide a useful framework for discussing process issues from a more global, and hopefully more quantitative, viewpoint.

Is Software Just an Assembly-Line Product?

By using the framework and concepts of API, it is now possible to discuss the issue of how assembly-line processes such as those used to make watches compare to processes used to make software. Figure 21 shows how a typical assembly-line process compares with a typical software process. Both diagrams in the figure should be interpreted on a "per product" basis — that is, the product unit for the assembly line process is one new bicycle, and the product unit for the software process is one new software product.

A conspicuous feature of the bicycle example — and indeed, of *any* assembly-line manufacturing process — is the very strong allocation wall that exists between parallel processes. Such a strong wall is possible simply because there is *no negotiation whatsoever* about the "design" of each new bike. All such allocations are made on the basis of a predefined "template" that gives exact, *unchanging* specifications for each bicycle. The overall configuration of the bicycle assembly line may eventually be changed, but on a "per product" basis, it remains very stable.

Another significant impact of the very high per-product stability of assembly line processes is that their allocator activities may be very minimal or even nonexistent. This is possible because only one "predefined" allocation template is needed to define *all* items produced by the assembly line. How those parts will fit together in the final integration activity is very precisely defined "up front," so that a complex allocator activity is not needed.

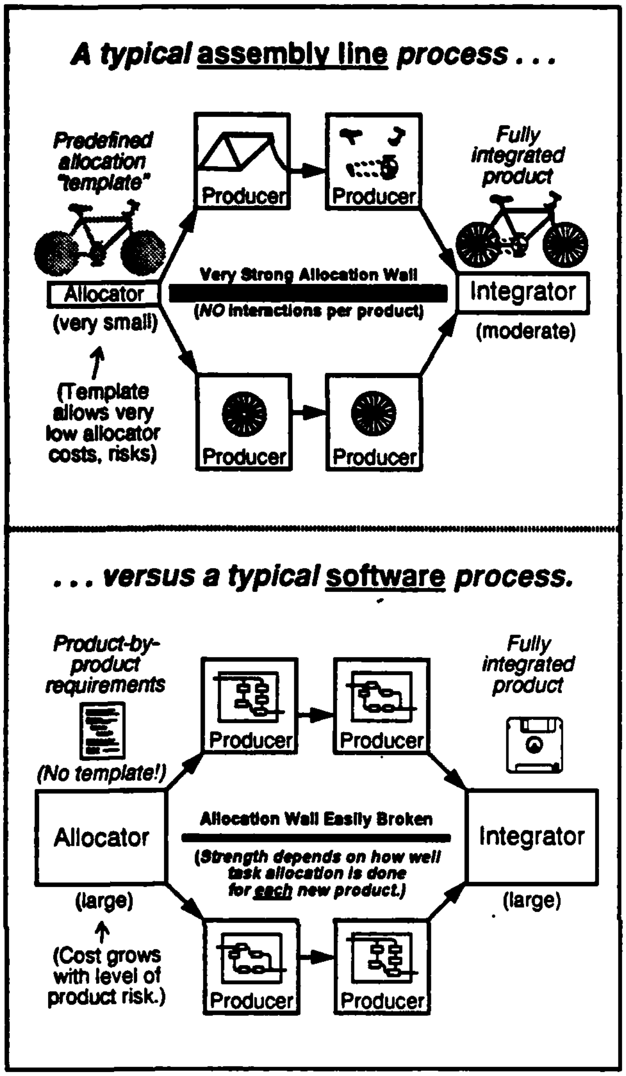

A typical assembly line process . . .
Predefined allocation "template"
Producer
Producer
Fully integrated product
Allocator
Very Strong Allocation Wall
(NO interactions per product)
Integrator
(very small)
(moderate)
(Template allows very low allocator costs, risks)
Producer
Producer
. . . versus a typical software process.
Product-by-product requirements
Fully integrated product
(No template!)
Producer
Producer
Allocator
Allocation Wall Easily Broken
(Strength depends on how well task allocation is done for each new product.)
Integrator
(large)
(large)
(Cost grows with level of product risk.)
Producer
Producer

Figure 21 - Assembly Lines Vs. Software Processes

The situation for a software process is quite different, since software *must* change product to product — otherwise, we would not normally call it software! Since the magnetic media folks solved the problem of *exact* replication of software products many years ago, software processes are now left with the much more difficult task of learning how to design a series of *similar* (but always distinct) software products.

The most important consequence of product variability in an API diagram is that it introduces risk into the structure of the process — specifically, an increased risk that allocation walls in the process will be violated. This in turn means that the allocator will need to be much more significant in both its size and its responsibilities, and that the producer activities should also work to "catch" internal inconsistencies at the earliest possible time. Integration also increases in importance, since it must in effect verify the earlier work of the allocator.

Now let's look a bit more closely at the implications of the "software is like watches" model of quality control. A critical feature of assembly line processes that allows them to work smoothly is the strength of their allocation walls, which allow each part of the process to "break out" into a very tightly defined process pipe. Intensive, product-only quality control works very well in such tightly defined pipes, since there is no significant danger of another process branch "coming in" and changing the basic definition of the product.

To make software follow this model, it will be necessary to create the same general scenario — a set of tightly defined process pipes that are separated by very strong allocation walls. There are two ways this might be done:

- Increase the size and cost of the allocator. The allocator can be given more resources with which to explore alternatives and verify work breakdowns. This approach is more common than one might think — it is more commonly called "prototyping." The main difference is that in the API framework prototyping is viewed more as a technique for developing strong allocation walls for the *process,* and less as a product-oriented technique. In many cases this distinction makes little difference, but there are situations where it can be significant.

- Decrease the complexity of the product. Although it sounds a bit strange, another way of making the assembly line analogy fit the software case more closely would be to greatly reduce product variability for a given software process. This is clearly a special case, since most "interesting" software problems to both the DoD and the software industry in general are not amenable to such specialization. Moreover, if a line of software products are *too* well defined it begins to be difficult to explain why the whole line is not replaced by an automated product configuration tool.

Neither of these two paths provides a fully satisfactory approach to the problem of how to build low-defect software predictably, but the first method *risk reduction* (Figure 22) would seem to come closer to handling the full range of software problems seen in industry. The second technique of *risk avoidance* (that is, of reducing process risks by focusing only on a very narrow range of software products) is useful only for specialized cases.

Interestingly enough, reducing process risks by reducing products variability is similar to software maintenance. In software maintenance, each "product" of the maintenance activity corresponds to a release of the (slightly) updated software product, and risk is automatically kept at lower levels by the fact that the majority of the code and its environment are quite stable.

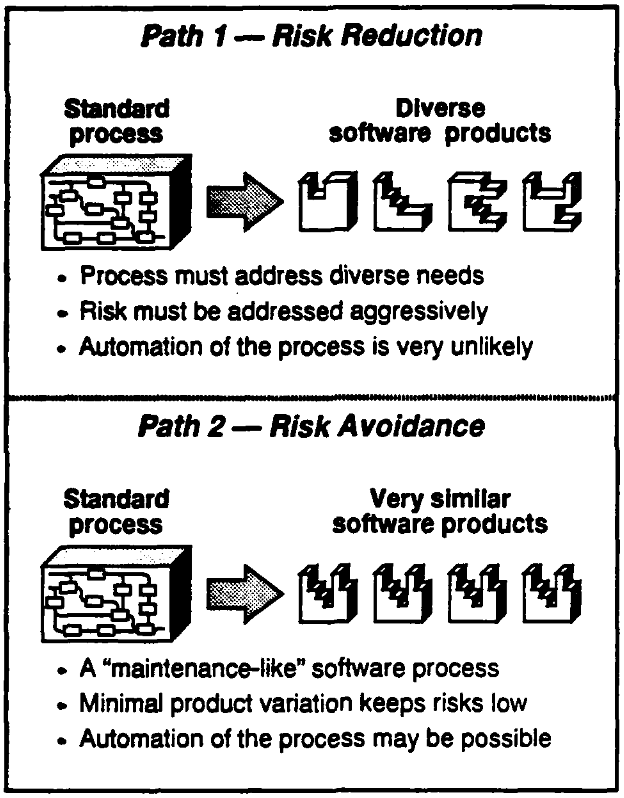

Path 1 — Risk Reduction
Standard process
Diverse software products
• Process must address diverse needs
• Risk must be addressed aggressively
• Automation of the process is very unlikely
Path 2 — Risk Avoidance
Standard process
Very similar software products
• A "maintenance-like" software process
• Minimal product variation keeps risks low
• Automation of the process may be possible

Figure 22 - Two Paths to Low-Defect Software

Conclusions

The SEI Process Assessment Program Is Great

For anyone who has read all the way through this paper, it may come as somewhat of a shock that the authors are both firmly convinced that SEI has made one of the truly outstanding contributions to software engineering in the past decade. In addition to building a superbly crafted, smoothly performing process assessment program, they have succeeded in getting the software industry to look seriously at the importance of understanding and controlling their software processes. Another SEI goal was to provoke industry comment on their model — and if this small paper is any indication, they have succeeded in that goal as well.

The only suggestion for the assessment program is that some type of structured, graphical method for recording process structures would be helpful for recording the kind of detailed process information available during assessments.

The SEI Evaluation Program Needs *Lots* of Work

If the SEI Assessment Program is the Mona Lisa of what SEI has done so far, then the SEI Capability Evaluation program is what one might get by applying a hole punch 85 times on that great picture and then claiming that the resulting 85 bits of paper on the floor somehow captured the "true beauty" of picture.

A full litany of the flaws in the SEI Capability Assessment program would be too long to list in this conclusion; the reader is directed back to the first half of the paper, and Figure 11 in particular, for all the sordid details. What we will instead list here is a few ideas for how SEI might avoid some of the traps they appear to have leaped into on the first go-round:

- Make the grading system is *totally* "up front."
- Make sure the grading system is statistically reliable.
- Avoid introducing artificial barriers that needlessly prevent an organization from making progress.
- Avoid grading structures in which missing the wrong two questions out of 85 results in an F.
- Admit that technology *is* an issue, and test for it.
- Try hard not to favor inefficient processes
- Make sure the test is broad enough to accommodate complex, *risky* design problems.
- Try to reward companies that eliminate processes or parts of processes through automation
- Stop treating software as though it were watches on an assembly line. It was not, is not, and never will be.
- Look for whether the process design matches the level of risk (variability) in the software product line.
- Develop a deeper understanding of how *processes* work, instead of focusing mostly on products.
- Consider dropping yes/no questionnaires altogether.
- Consider dropping *capability evaluations* altogether.

Level 5 Is Seriously Flawed

The specific methods described for Level 5 lead to very weak process improvement paradigm, one that is roughly equivalent to "redesigning" a program through the use of debugging methods only. Level 5 needs to be either very seriously reexamined or abandoned.

The Five-Level Model Is Flawed in Its Direction

The intent of the five-level SEI process maturity model is to take *design* intensive software organizations to new levels of quality and flexibility. The actuality is that the SEI model appears to strongly favor *maintenance* processes that have very narrow product definitions and few reasons to change. Elevating these types of effective but highly specialized organizations to the position of "industry standards" for *all* types of software development is likely to damage both the software industry and DoD's ability to contract for any type of new, highly complex software system.

Such disparities are possible primarily because the SEI model was never rigorously proven before its use in SEI evaluations. One need look no further than the main mirror of the Hubble Space Telescope to get an idea of the dangers of building an "ideal" testing template that turned out not to be ideal at all. The Hubble "experiment" with building a complex system around a flawed testing template was a very costly one, but how might it compare to the cost of redesigning the entire software *industry* around a "testing template" that may later prove to be seriously flawed?

The huge Hubble error could have been caught by making a few very simple "bottom line" checks about how well the mirror actually *worked,* as oppose to how well it matched an abstract test model. Perhaps there are simple tests that could also help SEI evaluate the actual "bottom line" quality and productivity performance of organizations that have used their maturity model.

The Fundamental "Assembly Line" Idea is Wrong

Software processes must deal with design changes *every time* they produce a product. Assembly lines do not deal with design changes in every product, so they can use a much simpler process model. SEI has their model backwards; assembly line processes are a subset of the software process, not vice-versa.

The consequences of this reversal of roles is that the SEI model persistently underestimates both the nature and complexity of the software problem.

Last Remarks

Physics provides the best model for understanding how one goes about analyzing very complex systems. Early physics was mostly a compilation of rules and facts with which some results could be predicted, and others not at all. These rules worked after a fashion, but any time a new system was discovered, its properties had to be analyzed anew.

Physics became a true science when it began to perceive *and prove* the existence of unifying themes, concepts that pulled many seemingly disparate pieces of information into a few orderly frameworks. The incomplete and entirely separate list of properties and behaviors for such diverse phenomena as light, heat, magnets, lightning, batteries, and radio waves began shifting and merging as new insights were gained, until finally a beautiful and incredibly powerful new framework of electromagnetism was born, a framework that allowed deep insights into the nature of the world around us and allowed us to build a world of technology and innovation.

Computer science is the fledgling science of information and information processing, and it is a field that the authors suspect will someday prove to be as full of deep insights as physics. At present we have our many little sets of rules and ideas and facts and figures, but we lack a way to tie them all together. A software process is just as much an aspect of information science as is the transmission of data over a noisy channel, but until we have perceived *and proven* the underlying themes that ties such disparate ideas into a unified framework, it is very unlikely that we will be able to specify exactly what the "best" features of a software process might truly be.

Acknowledgments

We would like to thank Bruce H. Barnes of the National Science Foundation for the many discussions in which he contributed to the ideas in this paper. We would also like to thank Thomas Reid and Shawn Bohner of the Contel Technology Center for their discussions and work that helped in the writing of this paper.

References

1. Watts S. Humphrey, "Characterizing the Software Process: A Maturity Framework." *IEEE Software,* March 1988, pp. 73-79.
2. Watts S. Humphrey, *Managing the Software Process.* Addison-Wesley, 1989.
3. Bruce Barnes and Terry Bolliner, "Making Reuse Cost Effective." *IEEE Software,* January 1991.
4. David A. Marca and Clement L. McGowan, *SADT — Structured Analysis and Design Technique.* McGraw-Hill, 1988.

Terry B. Bollinger is a senior member of the technical staff at the Contel technology Center. He works in the software-engineering laboratory on applied software research topics ranging from reuse and software maintenance to cost-oriented modeling of software processes. Bollinger has M.S. and B.S. degrees in computer science from the University of Missouri at Rolla. He is a member of the IEEE.

Dr. Clement McGowan is a Principal Scientist in the Software Engineering Laboratory of the Contel Technology Center (CTC) where he manages the Process and Metrics Project. Clem joined the CTC in May 1989 after six years as Manager of the Computer and Information Systems Department at GTE Laboratories where he served also on task forces dealing with software issues related to future intelligent networks. He co-authored the recent book SADT Structured Analysis and Design Technique (McGraw-Hill, 1988). From 1976 to 1983 Dr. McGowan was a Principal Consultant at SofTech, Inc. There he led project start-up, system analysis, and high level design using SADT for a wide range of applications. At SofTech Dr. McGowan also co-authored with Dr. Peter Freeman the video assisted instruction course Software Engineering and its Structured Methodologies (ASI, 1978). From 1969 to 1976 he was on the faculty of Brown University where he taught its first software engineering course in 1973. Dr. McGowan was also an Advisory Programmer at IBM Federal Systems Division with Dr. Harlan Mills' Advanced Programming Technology Group. Out of that experience he co-authored the book Top-Down Structured Programming Techniques (Petrocelli/Charter–now Van Nostrand, 1975). Dr. McGowan holds a Ph.D. in Computer Science from Cornell University.

Address questions about this article to either Terry Bollinger (Internet terry@ctc.contel.com) or Clem McGowan (Internet clem@ctc.contel.com), who both located at Contel Technology Center, 15000 Conference Center Drive, Chantilly, Virginia 22021-3808.

THE MYSTERY: WHY DO MANY OF THE VARIABLES DECLARED IN ADA PROGRAMS MODEL CONCEPTUALLY INVARIANT OBJECTS?

J. A. Perkins
Dynamics Research Corporation
Systems Division
60 Frontage Road, Andover, MA 01810

ABSTRACT

Our metric-driven analysis of Ada source for several military projects indicates that a high percentage of conceptually invariant objects are declared as variables, and that "run time" CONSTANTS are seldom if ever declared. More importantly, our analysis provides no evidence that programmers 1) are giving any consideration to whether the declared variables model conceptually variant or invariant objects, or 2) are being educated on how to specify conceptually invariant objects as CONSTANTs.

In this paper, we informally discuss what is meant by conceptual invariance and show examples of several programming techniques that support specifying invariant objects as CONSTANTs. Our research shows that use of these programming techniques would significantly reduce the number of variable declarations in each of the projects analyzed. We also examine specific instances where modeling conceptually invariant objects as CONSTANTs is impractical.

KEYWORDS

Software practices, software quality, Ada

1. INTRODUCTION

Our analysis of over 2 million text lines of Ada source for more than 20 military projects indicates that 1) very few "run time" CONSTANTs are declared in any of the projects analyzed, and 2) over 50% of the variables are modeling invariant objects. In fact, many of the projects contain no "run time" CONSTANTs and, in at least one instance, over 90% of the variable declarations model invariant objects. See [Levine90, Perkins89, Anderson88, Perkins87, Perkins86] for more detail on the analysis of these projects.

Our examination of those variable declarations modeling conceptually invariant objects reveals that, in most instances, there are practical techniques for replacing or eliminating such variable declarations. So, an obvious question is "Why do many of the variables declared in Ada programs model conceptually invariant objects, considering there exist practical techniques for specifying almost all such invariant objects as CONSTANTs?" This leads one to ask the more fundamental question, "Is it important (or how important is it) to capture as part of the declaration of an invariant object that the object is indeed invariant?"

In the opinion of the author, few programming practices affect the ease of determining the intended semantics of the software more than declaring invariant objects as CONSTANTs rather than as variables. Determining that an object has been assigned the proper value is easy when compared to the effort required to determine that this value remains unchanged until such time as the object is referenced. Knowing what objects are invariant is beneficial 1) when reasoning either formally or informally about the correctness of software, or 2) when deciphering (debugging) the intended operational semantics of incorrect software. The importance of ensuring the invariance of information when issues of security or safety are involved is emphasized in the research of the Byzantine generals' problem [Lamport82] and security models [Bell76]. Yet, our analysis indicates that programmers of mission-critical software are making no attempt to ensure that variables are not used to represent conceptually invariant objects.

In Ada, the default mode for parameters provides read-only access; surprisingly, this is not the case when declaring an object as either a variable or a constant. The only syntactic difference between a variable declaration with initialization and a constant declaration is the absence or presence of the keyword CONSTANT. Hence, read-write access, not read-only, can be viewed as the default for such declarations.

Although our analysis indicates that many conceptually invariant objects are declared as variables, our analysis also indicates that the mode of parameters requiring read-only access is usually defaulted or explicitly specified as IN. The mode IN OUT is seldom specified for such parameters. Write access to parameters is, for the most part, provided only when such access is needed. In fact, it is not unusual to find projects that have almost no CONSTANT declarations; yet all parameters requiring read-only access are explicitly specified as IN mode.

These findings lead one to ask the following, "If Ada required the explicit use of the keyword VARIABLE in variable declarations, would the number of variables used to model conceptually invariant objects be significantly reduced?" Granted, as the techniques in Section 3 of this paper illustrate, declaring conceptually invariant objects as constants instead of as variables is clearly more difficult than specifying the proper level of read-write access for parameters. Yet, when coding, this author forgets to include the keyword CONSTANT about 10 percent of the time when intending to declare an object as a constant.

In the remainder of this paper, we assume that there are advantages to capturing the variance or invariance of an object as part of the declaration of that object. As a result, our discussions focus on 1) informally defining what is meant by a variable modeling a conceptually invariant object and by a variable modeling multiple conceptual objects [Section 2], 2) practical techniques for eliminating variable declarations that model conceptually invariant objects, and the disadvantages, if any, of using these techniques [Section 3], and 3) specific instances where using CONSTANTS to model conceptually invariant objects is impractical [Section 4].

2. CONCEPTUALLY INVARIANT OBJECTS

Informally, we say that a variable models a conceptually invariant object when the variable is assigned, at most, one meaningful value during the life of the variable.

The life of a variable is the time span from allocation to deallocation of that variable. For example, examine the following code:

```
item_loop:
loop
  item_block:
  declare
    item_lv: item_type;
    -- A different variable, each named ITEM_LV,
    -- is allocated and deallocated on each
    -- iteration of ITEM_LOOP. The life of
    -- each of these variables is limited by
    -- the time required to execute the
    -- corresponding iteration of ITEM_LOOP.
  begin -- item_block
    -- ...
  end item_block;
end loop item_loop;
```

The variable declaration ITEM_LV: ITEM_TYPE inside declare block ITEM_BLOCK results in a different variable, each named ITEM_LV, for each iteration of loop ITEM_LOOP. The life of each of these variables, named ITEM_LV, is no longer than the time required to execute the corresponding iteration of loop ITEM_LOOP.

A value assigned to a variable is meaningful when that assigned value is read from the variable on at least one program path. For example, examine this second code sample:

```
function absolute
  (value_i: in value_type)
return nonnegative_value_type
is
  absolute_value_lv: absolute_value_type := 0;
  -- 0 is a meaningless value since this value
  -- is never read from the variable
  -- ABSOLUTE_VALUE_LV.
begin -- absolute
  -- absolute_value_if:
  -- The variable ABSOLUTE_VALUE_LV is assigned
  -- either the value represented by VALUE_I or
  -- -VALUE_I, but never more than one of these
  -- values on any call to FUNCTION ABSOLUTE.
  if value_i >= 0
  then
    absolute_value_lv := value_i;
    -- The value represented by VALUE_I is
    -- meaningful, since this value is read from
    -- the variable ABSOLUTE_VALUE_LV as part of
    -- the RETURN.
  else -- value_i < 0
    absolute_value_lv := -value_i;
    -- The value represented by -VALUE_I is
    -- meaningful, since this value is read from
    -- the variable ABSOLUTE_VALUE_LV as part of
    -- the RETURN.
  end if; -- absolute_value_if
  return absolute_value_lv;
  -- The value of variable ABSOLUTE_VALUE_LV
  -- is read as part of the RETURN.
end absolute;
```

In the above example, the value 0 assigned to variable ABSOLUTE_VALUE_LV during the allocation of ABSOLUTE_VALUE_LV is not meaningful, since this value is not read from ABSOLUTE_VALUE_LV on any program path. However, the values represented by VALUE_I and -VALUE_I assigned to ABSOLUTE_VALUE_LV inside the IF statement ABSOLUTE_VALUE_IF are meaningful, since these values are read fromABSOLUTE_VALUE_LV on at least one program path.

Furthermore, the variable ABSOLUTE_VALUE_LV models a conceptually invariant object, since during the life of each variable ABSOLUTE_VALUE_LV, ABSOLUTE_VALUE_LV is assigned at most one meaningful value, namely either the value represented by VALUE_I or the value represented by -VALUE_I. In other words, from the time the variable ABSOLUTE_VALUE_LV is allocated to the time the variable is deallocated, only a single value is assigned to the variable that is ever read from the variable. Thus, the variable ABSOLUTE_VALUE_LV models a conceptually invariant object. Notice that a different variable ABSOLUTE_VALUE_LV is allocated and deallocated on each call to function ABSOLUTE. The life of each variable ABSOLUTE_VALUE_LV corresponds to the time required to execute the function ABSOLUTE.

Informally, we say that a variable models multiple conceptual objects when there exists a time period between two assignments of meaningful values to the variable, where the variable could be deallocated and then reallocated without affecting the operational semantics of the program. We are assuming that the deallocation of a variable causes the value of that variable to be lost.

Examine the following example:

```
procedure echo
is
  item_lv: item_type;
begin -- echo
  item_loop:
  loop
    -- The variable ITEM_LV could be
    -- deallocated and reallocated at
    -- this point, since the current
    -- value of ITEM_LV has no effect
    -- on the value assigned to ITEM_LV
    -- by GET.
    get(item_lv);
    put(item_lv);
  end loop item_loop;
end echo;
```

Here, the variable ITEM_LV models multiple conceptual objects since the variable ITEM_LV could be deallocated and reallocated at the beginning of each iteration of the loop ITEM_LOOP without affecting the operational semantics of procedure ECHO.

3. METHODS FOR ELIMINATING VARIABLES

A high percentage of the variable declarations used to model conceptually invariant objects can be eliminated or replaced by constant declarations, by using one or more of the following techniques:

Eliminating the variable declaration by substituting the expression used to initialize the variable in place of the (singular) read of that variable [Section 3.1],

Changing the variable declaration to a constant declaration by simply adding the keyword CONSTANT [Section 3.2],

Creating a local block to delay the declaration of the object until the value of the object is able to be computed [Section 3.3],

Converting the PROCEDURE that computes the value to a FUNCTION, so that the computed value can be used as part of the declaration of the object [Section 3.4],

Creating a local FUNCTION that computes the value in those instances where the value to be assigned is conditional [Section 3.5],

Converting the LOOP that computes the value to a FUNCTION, so that the computed value can be used as part of the declaration of the object [Section 3.6], or

Unrolling a LOOP, so that the only value assigned to an object is assigned outside the LOOP instead of assigned on the "first" or "last" iteration of that LOOP [Section 3.7].

Each of the above techniques involving creation of a FUNCTION is useful for providing initialization as part of a variable declaration, even when the variable is modeling a conceptually variant object.

The subparagraphs to follow illustrate the use of these programming techniques. The examples of variables modeling invariance shown in these subparagraphs are indicative of those encountered in our analysis of actual Ada source.

3.1 ELIMINATE THE DECLARATION

Our analysis indicates that it is not unusual to find conceptually invariant objects declared as variables that are read only once or not read at all.

Clearly, variable declarations for variables that are never read can be eliminated. The example source below illustrates how to eliminate a variable declaration if the corresponding variable is written to only once and read only once.

```
Original Code:

function number_of_roots
  (a_coefficient_i: in positive_coefficient_type;
   b_coefficient_i: in coefficient_type;
   c_coefficient_i: in coefficient_type)
return number_of_roots_type
is
  discriminant_lv: coefficient_type
    :=  b_coefficient_i*b_coefficient_i
       - 4*a_coefficient_i*c_coefficient_i;
  -- This is the only place where a value
  -- is assigned to the variable
  -- DISCRIMINANT_LV.
begin -- number_of_roots
  -- number_of_roots_case:
  case discriminant_lv is
  -- This is the only place where the value
  -- of the variable DISCRIMINANT_LV is read.
  when positive_coefficient_type =>
    return two_roots;
  when zero_coefficient_type =>
    return one_root;
  when negative_coefficient_type =>
    return zero_roots;
  end case; -- number_of_roots_case
end number_of_roots;
```

```
Modified Code:

function number_of_roots
  (a_coefficient_i: in positive_coefficient_type;
   b_coefficient_i: in coefficient_type;
   c_coefficient_i: in coefficient_type)
return number_of_roots_type
is
begin -- number_of_roots
  -- number_of_roots_case:
  case   b_coefficient_i*b_coefficient_i
       - 4*a_coefficient_i*c_coefficient_i
  -- Substitute the expression used to
  -- initialize the variable DISCRIMINANT_LV
  -- in place of that variable.
  is
  when positive_coefficient_type =>
    return two_roots;
  when zero_coefficient_type =>
    return one_root;
  when negative_coefficient_type =>
    return zero_roots;
  end case; -- number_of_roots_case
end number_of_roots;
```

In the modified code above, the expression used to initialize the variable DISCRIMINANT_LV (which is the only place where this variable is assigned a value) in the original code is merely substituted for the only occurrence where the contents of the variable DISCRIMINANTs is read, namely in the expression of the CASE statement. Since this expression for calculating the discriminant is only evaluated once in each version of the FUNCTION NUMBER_OF_ROOTS, there will be no negative impact on efficiency.

This method of substituting the initializing expression for the singular read of the variable is the degenerate case of the method illustrated in Section 3.2. This method of substitution can also be used to eliminate constant declarations.

3.2 SIMPLY ADD THE KEYWORD CONSTANT

Our analysis indicates that it is not unusual to find conceptually invariant objects declared as variables that could be declared as constants simply by adding the keyword CONSTANT to those declarations. This occurs quite frequently when 1) the invariant object is declared using an access type, or 2) the value assigned to the invariant object is calculated in terms of global variables, parameters, or other local variables.

In the original code of the example below, the invariant objects declared as variables, namely DISCRIMINANT_LV, TWO_COEFFICIENT_A_LV, and SINGLE_ROOT_LV are assigned values based on input parameters and/or other local variables. The associated modified code illustrates how easily these variables can be declared as CONSTANTs. Using this simple method, many of the variable declarations encountered during our analysis could be modified to be constant declarations without affecting the operational semantics of the source.

```
Original Code:

function roots
  (a_coefficient_i: in positive_coefficient_type;
   b_coefficient_i: in coefficient_type;
   c_coefficient_i: in coefficient_type)
return roots_type
is
  discriminant_lv: coefficient_type
    :=   b_coefficient_i*b_coefficient_i
       - 4*a_coefficient_i*c_coefficient_i;
  -- This is the only place where a value
  -- is assigned to the variable
  -- DISCRIMINANT_LV.
  two_coefficient_a_lv: coefficient_type
    :=   2 * coefficient_a_i;
  -- This is the only place where a value is
  -- assigned to the variable
  -- TWO_COEFFICIENT_A_LV.
begin -- number_of_roots
  -- roots_if:
  if     discriminant_lv
     in positive_coefficient_type
  then -- root_if
    return roots_type'
      (the_number_of_roots => two_roots,
       the_first_root  =>
          (  b_coefficient_i
           + square_root(discriminant_lv))
         / two_a_coefficient_lv,
       the_second_root =>
          (  b_coefficient_i
           - square_root(discriminant_lv))
```

```
        / two_a_coefficient_lv);
  elsif    discriminant_lv
        in zero_coefficient_type
  then -- root_if
    single_root_block:
    declare
      single_root_lv: coefficient_type
        :=  b_coefficient_i
           / two_a_coefficient_lv;
      -- This is the only place where a value
      -- is assigned to the variable
      -- SINGLE_ROOT_LV.
    begin -- single_root_block
      return roots_type'
        (the_number_of_roots => one_root,
         the_first_root  => single_root_lv,
         the_second_root => single_root_lv);
    end single_root_block;
  elsif    discriminant_lv
        in negative_coefficient_type
  then -- root_if
    return roots_type'
      (the_number_of_roots => zero_roots,
       the_first_root  => undefined_root,
       the_second_root => undefined_root);
  end if; -- root_if
end roots;
```

Modified Code:

```
function roots
  (a_coefficient_i: in positive_coefficient_type;
   b_coefficient_i: in coefficient_type;
   c_coefficient_i: in coefficient_type)
return roots_type
is
  discriminant_lc: constant coefficient_type
    :=   b_coefficient_i*b_coefficient_i
       - 4*a_coefficient_i*c_coefficient_i;
  -- Simply add the keyword CONSTANT to indicate
  -- that the object DISCRIMINANT_LC remains
  -- invariant.
  two_coefficient_a_lc: constant
    coefficient_type
    :=   2 * coefficient_a_i;
  -- Simply add the keyword CONSTANT to indicate
  -- that the object TWO_COEFFICIENT_LC remains
  -- invariant.
begin -- number_of_roots
  -- roots_if:
  if    discriminant_lc
    in positive_coefficient_type
  then -- root_if
    return roots_type'
      (the_number_of_roots => two_roots,
       the_first_root  =>
           (  b_coefficient_i
            + square_root(discriminant_lc))
          / two_a_coefficient_lc,
       the_second_root =>
           (  b_coefficient_i
            - square_root(discriminant_lc))
          / two_a_coefficient_lc);
  elsif    discriminant_lc
        in zero_coefficient_type
  then -- root_if
    single_root_block:
    declare
      single_root_lc: constant coefficient_type
        :=   b_coefficient_i
            / two_a_coefficient_lc;
      -- Simply add the keyword CONSTANT to
      -- indicate that the object SINGLE_ROOT_LC
      -- remains invariant.
    begin -- single_root_block
      return roots_type'
        (the_number_of_roots => one_root,
         the_first_root  => single_root_lc,
         the_second_root => single_root_lc);
    end single_root_block;
  elsif    discriminant_lc
        in negative_coefficient_type
  then -- root_if
    return roots_type'
      (the_number_of_roots => zero_roots,
       the_first_root  => undefined_root,
       the_second_root => undefined_root);
  end if; -- root_if
end roots;
```

When access types are used, variables are often used to model the pointer object, even though the address pointed to by that object never changes. In the example below, the values of the objects pointed to change, but each of the pointer variables HEATER_LV, THERMOMETER_LV, and DISTURBANCE_LV is always pointing to the same object. Again, the variables can be changed to constants by merely adding the keyword CONSTANT to each variable declaration.

Original Code:

```
procedure simulate
is
  heater_lv: heater_type
    := new heater_node_type;
  -- The variable HEATER_LV points to the
  -- same heater object for the entire
  -- simulation.
  thermometer_lv: thermometer_type
    := new thermometer_node_type;
  -- The variable THERMOMETER_LV points to the
  -- same thermometer object for the entire
  -- simulation.
  disturbance_lv: disturbance_type
    := new disturbance_node_type;
  -- The variable DISTURBANCE_LV points to the
  -- same disturbance object for the entire
  -- simulation.
begin -- simulate
  initialize
    (the_header_i => heater_lv,
     the_thermometer_i => thermometer_lv,
     the_disturbance_i => disturbance_lv);
  schedule(the_event_i => header_lv);
  schedule(the_event_i => thermometer_lv);
  schedule(the_event_i => disturbance_lv);
  run;
end simulate;
```

Modified Code:

```
procedure simulate
is
  heater_lc: constant heater_type
    := new heater_node_type;
  -- Simply add the keyword CONSTANT to indicate
  -- that HEATER_LC always points to the same
  -- object.
  thermometer_lc: constant thermometer_type
    := new thermometer_node_type;
  -- Simply add the keyword CONSTANT to indicate
  -- that THERMOMETER_LC always points to the
  -- same object.
  disturbance_lc: constant disturbance_type
    := new disturbance_node_type;
  -- Simply add the keyword CONSTANT to indicate
  -- that DISTURBANCE_LC always points to the
  -- same object.
begin -- simulate
  initialize
    (the_header_i => heater_lc,
     the_thermometer_i => thermometer_lc,
     the_disturbance_i => disturbance_lc);
  schedule(the_event_i => header_lc);
  schedule(the_event_i => thermometer_lc);
  schedule(the_event_i => disturbance_lc);
  run;
end simulate;
```

Notice that the changes from variable declarations to constant declarations in the above examples have no negative impact on efficiency.

In the modified code examples above, each of the constants, DISCRIMINANT_LC, TWO_A_COEFFICIENT_LC, SINGLE_ROOT_LC, HEATER_LC, THERMOMETER_LC, and DISTURBANCE_LC, is an example of a "run time" CONSTANT. The values for these constants are not known until run time and the values can be different for each allocation of these constants.

"Run time" constants seldom appeared in any of the code we analyzed, even for situations as straightforward as those above. Most traditional Von Neumann languages support constant declarations only when the values used to initialize those constants are known at compile time. In other words, the initializing expressions for constants in these other languages must be static. This leads one to question whether the current text books, training courses, programming standards, and style guidelines on Ada have been lax concerning educating programmers on Ada's support for "run time" constants. Certainly, the topic of "run time" constants in Ada has been given little emphasis by authors such as Booch, Barnes, and Software Productivity Consortium [Booch83, Barnes89, SPC89].

3.3 DELAY THE DECLARATION

Our analysis indicates that many conceptually invariant objects declared as variables could be declared as constants by merely delaying the declaration of the invariant object until such time as the value used to initialize the object is available.

In the original code of the example below, the variable FRAME_LV models a conceptually invariant object. Delaying the declaration of this object until after the value of the variable POSITION_LV is available allows the object to be declared as a constant. In the modified code, the block DECLARE_FRAME_LC_BLOCK is placed after the call to procedure GET_POSITION and the constant FRAME_LC is declared in this block. The impact of this change on efficiency is negligible.

Original Code:

```
procedure display_frame
is
  position_lv: position_type;
  frame_lv: frame_type;
begin -- display_frame
  get_position(position_file,position_l\
  frame_lv := make_frame(position_lv);
  -- This is the only place where a value
  -- is assigned to the variable FRAME_LV.
  -- ...
end display_frame;
```

Modified Code:

```
procedure display_frame
is
  position_lv: position_type;
begin -- display_frame
  get_position(position_file,position_lv);
  declare_frame_lc_block:
  declare
    frame_lc: constant frame_type
      := make_frame(position_lv);
    -- Delaying the declaration allows
    -- FRAME_LC to be declared as a
    -- CONSTANT.
  begin -- declare_frame_lc_block
    -- ...
  end declare_frame_lc_block;
end display_frame;
```

3.4 USE FUNCTION TO REPLACE PROCEDURE

Our analysis indicates that many conceptually invariant objects declared as variables could be declared as constants by changing the PROCEDURE used to initialize the object to a FUNCTION, allowing the initialization to be part of the declaration of that object.

In the original code of the example below, the variable POSITIONAL_LV models a conceptually invariant object. Changing the procedure GET_POSITION of the original version to the function GET_POSITION_SE in the modified version allows the invariant object to be declared as the constant POSITIONAL_LC.

```
Original Code:

procedure display_frame
is
  position_lv: position_type;
begin -- display_frame
  get_position(position_file,position_lv);
  -- This is the only place where a value is
  -- assigned to the variable POSITION_LV.
  declare_frame_lc_block:
  declare
    frame_lc: constant frame_type
      := make_frame(position_lv);
  begin -- declare_frame_lc_block
    -- ...
  end declare_frame_lc_block;
end display_frame;
```

```
Modified Code:

procedure display_frame
is
  position_lc: position_type
    := get_position_se(position_file);
  -- Changing the procedure GET_POSITION to
  -- the function GET_POSITION_SE allows
  -- POSITION_LC to be declared as a CONSTANT.
  -- The suffix _SE is used to indicate
  -- that the function GET_POSITION_SE causes
  -- a side-effect, namely the movement to
  -- the next record in file POSITION_FILE.
begin -- display_frame
  declare_frame_lc_block:
  declare
    frame_lc: constant frame_type
      := make_frame(position_lc);
  begin -- declare_frame_lc_block
    -- ...
  end declare_frame_lc_block;
end display_frame;
```

Although the conversion from a PROCEDURE to a FUNCTION has negligible impact on efficiency, changes of this kind often result in FUNCTIONs with side-effects. The trade-offs involved in deciding between declaring invariant objects as variables or providing FUNCTIONs with side-effects are many. The use of FUNCTIONs with side-effects in expressions involving more than one operation can result in an erroneous Ada program. The behavior of such programs is undefined.

We recommend providing both the PROCEDURE and the FUNCTION with side-effects, and then limiting the use of these kinds of FUNCTIONs to situations where initialization as part of a declaration is involved.

3.5 USE FUNCTION TO REPLACE IF

Our analysis indicates that some conceptually invariant objects declared as variables could be declared as constants by moving the IF statement or CASE statement containing the initialization of the object to a FUNCTION,thereby allowing theinitialization to be part of the declaration of that object.

In the original code of the example below, the variable STATUS_LV models a conceptually invariant object. The CASE statement RECEIVE_STATUS_CASE contains the only instances where the variable STATUS_LV is assigned values, and then only one assignment per WHEN. In the modified version, a function RECEIVE_STATUS is created, and the CASE statement RECEIVE_STATUS_CASE is moved to this function. The constant STATUS_LC is initialized using this function.

```
Original Code:

procedure display_status
  (station_i: in station_type)
is
  status_lv: status_type;
  -- STATUS_LV is only assigned a value
  -- inside the RECEIVE_STATUS_CASE.
begin -- display_status
  -- receive_status_case:
  case station_i is
  when satellite =>
    status_lv := satellite.receive(station_i);
  when ground =>
    status_lv := ground.receive(station_i);
  end case; -- receive_status_case
  if status_lv = off_line
  then
    report_off_line(station_i);
  end if;
  -- ...
end display_status;
```

```
Modified Code:

function receive_status
  (station_i: in station_type)
return status_type
is
begin -- receive_status
  -- receive_status_case:
  case station_i is
  when satellite =>
    return satellite.receive(station_i);
  when ground =>
    return ground.receive(station_i);
  end case; -- receive_status_case
end receive_status;
```

```
procedure display_status
  (station_i: in station_type)
is
  status_lc: constant status_type
    := receive_status(station_i);
  -- Creating the function RECEIVE_STATUS allows
  -- STATUS_LC to be declared as a CONSTANT.
begin -- display_status
  if status_lc = off_line
  then
    report_off_line(station_i);
  end if;
  -- ...
end display_status;
```

The minor impact to efficiency of an extra FUNCTION call or to code space of an extra FUNCTION is, in most instances, more than offset by the increased modularity and the ability to specify the conceptual invariant as a CONSTANT.

3.6 USE FUNCTION TO REPLACE LOOP

Our analysis indicates, in a few instances, that conceptually invariant objects declared as variables could be declared as CONSTANTs by moving to a FUNCTION 1) the initializing assignment for the object, and 2) the LOOP statement performing at most a single assignment for the object, thereby allowing the initialization to be part of the declaration of that object. This is a special case of the conditional assignment shown in SECTION 3.5, where the LOOP statement acts as the then clause and the initializing assignment acts as the else clause.

In the original code of the example below, the variable TABLE_INDEX_LV is initially assigned the value 0 and then is assigned the value of the loop control variable INDEX if ELEMENT_I is found in TABLE. In other words, conceptually the variable TABLE_INDEX_LV is conditionally assigned 1) the value of the index corresponding to the element if the element is found, or 2) the value 0 if the element is not found. In the modified code, the function SEARCH is created, and the initial assignment statement and the loop SEARCH_LOOP are moved to this FUNCTION. In the function SEARCH, the assignment statement is replaced by a RETURN statement. The constant TABLE_INDEX_LC is initialized using this function.

```
Original Code:

procedure locate
  (element_i: in element_type)
is
  table_index_lv: extended_table_index_type;
  -- TABLE_INDEX_LV is assigned the value of
  -- INDEX into TABLE if ELEMENT_I is found, or
  -- 0 if ELEMENT_I is not found.
begin -- locate
  table_index_lv := 0;
  search_loop:
  for index in table_index_type
  loop
    if table(index) = element_i
    then
      table_index_lv := index;
      exit search_loop;
    end if;
  end search_loop;
  if table_index_lv = 0
  then
    -- ...
  end if;
end locate;

Modified Code:

function search
  (element_i: in element_type)
return extended_table_index_type
is
begin -- search
  search_loop:
  for index in table_index_type
  loop
    if table(index) = element_i
    then
      return index;
    end if;
  end search_loop;
  return 0;
end search;


procedure locate
  (element_i: in element_type)
is
  table_index_lc: constant
    extended_table_index_type
    := search(element_i);
  -- Creating the function SEARCH allows
  -- TABLE_INDEX_LC to be declared as a
  -- CONSTANT.
begin -- locate
  if table_index_lc = 0
  then
    -- ...
  end if;
end locate;
```

Again, the minor impact to efficiency of an extra FUNCTION call or to code space of an extra FUNCTION is, in most instances, more than offset by the increased modularity and the ability to specify the conceptual invariant as a CONSTANT.

Notice that our informal definition of conceptual invariance is not satisfactory to cover the conceptual invariance in the example above. Both of the values assigned to the variable TABLE_INDEX_LV are meaningful and both of these values are assigned to TABLE_INDEX_LV any time ELEMENT_I is found in TABLE. However, the value 0 is only read from the variable TABLE_INDEX_LV if ELEMENT_I is not found in TABLE.

3.7 LOOP UNROLLING

Our analysis indicates that some conceptually invariant objects declared as variables could be declared as CONSTANTs by moving the assignment of the object outside the LOOP in those instances where that assignment is always associated with either the first or last iteration of the LOOP. Usually when this practice is used, the practice of delaying the declaration of the object must also be used.

In the original code of the example below, the variable SIGN_LV is assigned a value inside the loop CONVERT_LOOP only on the first iteration of that loop, if at all. In the modified version, removal of the calculation of the sign from the loop CONVERT_LOOP and the creation of block CONVERT_BLOCK allows SIGN_LC to be declared as a CONSTANT. This also eliminates the need for the flag SIGN_FOUND_LV.

Original Code:

```
function convert return value_type
is
  char_lv: character;
  -- CHAR_LV models multiple conceptual
  -- objects, each of which is invariant.
  sign_lv: sign_type := plus;
  -- SIGN_LV is assigned a value on the
  -- first iteration of CONVERT_LOOP,
  -- if at all.
  sign_found_lv: boolean := false;
  -- SIGN_FOUND_LV is assigned the value
  -- true at the end of the first iteration
  -- of CONVERT_LOOP, and the value of
  -- SIGN_FOUND_LV remains true from then
  -- on.
  value_lv: value_type := 0;
  -- VALUE_LV represents a conceptually
  -- variant object.
begin -- convert
  convert_loop:
  loop
    exit when end_of_file(infile);
    get_char(infile,char_lv);
    if char_lv = "+"
    then
      if sign_found_lv
      then
        raise format_exception;
      end if;
      sign_lv := plus;
    elsif char_lv = "-"
    then
      if sign_found_lv
      then
        raise format_exception;
      end if;
      sign_lv := minus;
    else
      value_lv :=   10 * value_lv
                  + digit(char_lv);
    end if;
    sign_found_lv := true;
  end loop convert_loop;
  return sign_lv * value_lv;
end convert;
```

Modified Code:

```
function convert return value_type
is
begin -- convert
  if end_of_file(infile)
  then
    return 0;
  end if;
  convert_block:
  declare
    sign_char_lc: constant character
      := get_char_se(infile);
    -- Delaying the declaration allows
    -- SIGN_CHAR_LC to be declared as
    -- a CONSTANT.
    sign_lc: constant sign_type
      := what_sign(sign_char_lv);
    -- Delaying the declaration allows
    -- SIGN_LC to be declared as
    -- a CONSTANT.
    value_lv: value_type
      := digit(sign_char_lv);
    -- VALUE_LV represents a conceptually
    -- variant object. Inside CONVERT_LOOP,
    -- its current value depends on its
    -- previous value.

 begin -- convert_block
   convert_loop:
   loop
     exit when end_of_file(infile);
     declare_char_lc_block:
     declare
       char_lc: constant character
         := get_char_se(infile);
       -- Delaying the declaration allows
       -- CHAR_LC to be declared as
       -- a CONSTANT.
     begin -- declare_char_lv_block
       if    char_lc = "+"
          or char_lc = "-"
       then
         raise format_exception;
       else
         value_lv :=   10 * value_lv
                     + digit(char_lc);
       end if;
     end declare_char_lc_block;
   end loop convert_loop;
   return sign_lc * value_lv;
 end convert_block;
end convert;
```

The negative impact on code space is prohibitive only in those instances where large amounts of source code are associated with each iteration of the original LOOP. Use of this method often results in an increase in the number of branch statements.

Again, our informal definition of conceptual invariance is not satisfactory to cover the conceptual invariance in the example above. All of the values assigned to the variable SIGN_LV are meaningful and more than one of these values are assigned to SIGN_LV any time a sign appears as the first character in the input. However, the value PLUS assigned in the declaration of SIGN_LV is only read from the variable SIGN_LV if no sign appears in the input.

Notice that the variable CHAR_LV in the original version models multiple conceptual objects, each of which is invariant. By converting the procedure GET_CHAR to the function GET_CHAR_SE, and delaying the declarations of SIGN_CHAR_LC and CHAR_LC, each of these objects is declared as a CONSTANT. The variable VALUE_LV is conceptually variant, since the current value of this variable depends on its previous value.

4. WHEN CONSTANTS ARE IMPRACTICAL

In Ada, not all variable declarations used to model conceptually invariant objects can be eliminated or replaced by CONSTANT declarations. It is often impractical, if not impossible, to declare a conceptually invariant object as a CONSTANT any time that object is assigned its only value inside a structured code segment, such as a block statement, accept statement, or subprogram statement, and that object is read outside the scope of that structured code segment. In other words, declaring a conceptually invariant object in a CONSTANT declaration is often impractical whenever the scope of the object is more global than the structured code segment where the initialization of the object is performed.

Some of the later examples in Section 3 successfully converted variables modeling conceptually invariant objects requiring "extended" scopes to CONSTANTs. However, the example source below, which is explained in detail in [Barnes89], illustrates how a variable is required whenever a conceptually invariant object initialized inside an ACCEPT statement is read outside the ACCEPT statement. Declaring the conceptually invariant object inside the ACCEPT statement is impractical, since this forces the outside actions on the object to also be moved inside the ACCEPT statement, thereby greatly lengthening the time required for each rendezvous. Copying the variable into a CONSTANT after the ACCEPT statement offers no advantages, since the variable is still visible whenever the CONSTANT is visible. In [Perkins91], the advantages of assigning the value of the variable to a constant and then hiding the variable with a homograph are addressed.

In both the original code and the modified code, the variable PRIME_LV, although modeling an invariant object, cannot be declared as a CONSTANT, since the single assignment for this variable appears inside the first ACCEPT statement and then is read several times outside the scope of this ACCEPT statement. In the original code, the variable POTENTIAL_PRIME_LV models multiple conceptual objects, each of which is conceptually invariant. Moving the declaration of the variable POTENTIAL_PRIME_LV to a block inside the LOOP FIND_NEXT_OR_POTENTIAL_PRIME_LOOP would indicate that the current value assigned to the variable does not depend on the value in the previous iteration. However, the variable POTENTIAL_PRIME_LV could still not be declared as a constant because of the initialization of the variable POTENTIAL_PRIME_LV inside the second ACCEPT statement and the use of the variable outside the scope of this ACCEPT statement. In the modified code, the variables NEXT_PRIME_LV and POTENTIAL_PRIME_LV, which are used in place of the variable POTENTIAL_PRIME_LV in the original version, are both declared inside a block in their corresponding LOOPs, namely FIND_NEXT_PRIME_LOOP and FIND_POTENTIAL_PRIME_LOOP.

```
Original Code:

task body filter
  prime_lv: prime_type := undefined_prime;
  -- The only meaningful value for PRIME_LV
  -- is assigned inside the first ACCEPT.
  potential_prime_lv: prime_type
    := undefined_prime;
  -- The only meaningful values for PRIME_LV
  -- are assigned inside the second ACCEPT.
  here_lv: position_type := undefined_position;
  -- The only meaningful value for HERE_LV is
  -- assigned after immediately after the first
  -- ACCEPT.
  next_lv: a_filter_type;
  -- The only truly meaningful value for NEXT_LV
  -- is assigned once inside the inner IF after
  -- the second ACCEPT.
begin -- filter
  accept input(number: in prime_type) do
    prime_lv := number;
  end input;
  here_lv := make_frame(prime_lv);
  find_next_or_potential_prime_loop:
  loop
    accept input(number: in prime_type) do
      potential_prime_lv := number;
    end input;
    write_to_frame(potential_prime_lv, here_lv);
    -- potential_prime_to_next_filter_if:
    if potential_prime_lv mod prime_lv /= 0
    then
      -- next_prime_filter_task_if:
      if next_lv = null
      then
        next_lv := next filter_type;
      end if; -- next_prime_filter_task_if
      next_lv.input(potential_prime);
    end if; -- potential_prime_to_next_filter_if
    clear_frame(here_lv);
  end find_next_or_potential_prime_loop;
end filter;
```

```
Modified Code:

task body filter
  prime_lv: prime_type := undefined_prime;
begin -- filter
  accept input(number: in prime_type) do
    prime_lv := number;
  end input;
  declare_here_block_lc:
  declare
    here_lc := constant make_frame(prime_lv);
    -- Delaying the declaration allows
    -- HERE_LC to be declared as a CONSTANT.
  begin -- declare_here_lc_block
    find_next_prime_loop:
    loop
      declare_next_prime_lv_block:
      declare
        next_prime_lv: prime_type
          := undefined_prime;
        -- Delaying the declaration does not
        -- allow NEXT_PRIME_LV to be declared
        -- as a CONSTANT because of the
        -- initialization of this object inside
        -- an ACCEPT and the use of the object
        -- outside the ACCEPT.
      begin -- declare_next_prime_lv_block
        accept input(number: in prime_type) do
          next_prime_lv := number;
        end input;
        write_to_frame(next_prime_lv, here_lv);
        -- next_prime_filter_task_if:
        if next_prime_lv mod prime_lv /= 0
        then
          declare next_lc_block:
          declare
            next_lc: constant a_filter_type
              := new filter_type;
          -- Unrolling the loop and delaying
          -- the declaration allows NEXT_LC
          -- to be declared as a CONSTANT.

          begin -- next_lc_block
            next.input_lc(next_prime_lv);
            clear_frame(here_lc);
            find_potential_prime_loop:
            loop
              potential_prime_lv_block:
              declare
                potential_prime_lv: prime_type
                  := undefined_prime;
                -- Delaying the declaration does
                -- not allow POTENTIAL_PRIME_LV
                -- to be declared as a CONSTANT
                -- because of the initialization
                -- of this object inside an
                -- ACCEPT and the use of the
                -- object outside the ACCEPT.
              begin -- potential_prime_lv_block
                accept input
                  (number: in prime_type) do
                 potential_prime_lv := number;
                end input;
                write_to_frame
                  (potential_prime_lv,
                   here_lv);
                potential_prime_if:
                if    (    potential_prime_lv
                       mod prime_lv)
                   /= 0
                then
                  next_lc.input
                    (potential_prime_lv);
                end if; -- potential_prime_if
                clear_frame(here_lc);
              end potential_prime_lv_block;
            end find_potential_prime_loop;
          end declare next_lc_block;
        end if; -- next_prime_filter_task_if
        clear_frame(here_lc);
      end declare_next_prime_lv_block;
    end find_next_prime_loop;
  end declare_here_lc_block;
end filter;
```

The other two variables, HERE_LV and NEXT_LV declared in the original version of task FILTER, are also modeling conceptually invariant objects. Actually, under a loose interpretation of the informal definitions, the variable NEXT_LV models multiple conceptual objects, each of which is conceptually invariant. The value NULL, which is implicitly assigned to the variable NEXT_LV when the variable is allocated, merely acts as a means for checking whether this variable has been assigned its only "real" value, namely the next spawning of task FILTER.

In the modified version, the variable HERE_LC is declared as a CONSTANT by merely creating an inner block DECLARE_HERE_LC_BLOCK after the first ACCEPT statement. The variable NEXT_LC is declared as a CONSTANT by creating the block DECLARE_NEXT_LC_BLOCK and the loop FIND_POTENTIAL_PRIME_LOOP inside the IF statement NEXT_PRIME_FILTER_TASK_IF of the loop FIND_NEXT_PRIME_LOOP.

The source below illustrates another example where declaring a conceptually invariant object as a CONSTANT is impractical. In this case, the variable TABLE_LV is initialized inside the executable portion of the PACKAGE body of TABLE_PACKAGE. The value of the variable TABLE_LV is never written to by any of the subprograms inside this PACKAGE. However, it is read by all of these subprograms. The inefficiency resulting from needlessly recopying the values read from file TABLE_FILE prevents including a call to an initializing FUNCTION in the declaration of the object TABLE_LV.

```
Original Code:

package body table_package is
  table_lv: table_type;
  -- Implementation of the
  -- TABLE_PACKAGE subprograms.
  -- ...
begin -- table_package
  initialize_table_loop:
  for table_index in max_table_index_type
  loop
    exit when end_of_file(table_file);
    table_lv(table_index) := get_se(table_file);
  end initialize_table_loop;
end table_package;
```

In the above example, the use of an ACCESS TYPE in representing TABLE_TYPE could allow the object TABLE_LV to be declared as a CONSTANT. However, this would not prevent unintentional modification TABLE_PACKAGE by the subprograms contained in package of :he contents pointed to by TABLE_LV.

5. CONCLUSIONS

Our analysis of Ada source for several military projects indicates that a high percentage of conceptually invariant objects are declared as variables. Our research indicates that practical techniques do exist for declaring most conceptually invariant objects as CONSTANTs, thereby significantly reducing the number of variable declarations contained in the Ada source.

Although these techniques cannot eliminate the occasional need to declare a conceptually invariant object as a variable, their use results in most variables representing either 1) permanent storage locations having values that change over time, such as for stacks, queues, and graphs; or 2) temporary storage locations for accumulating data inside of loops, such as for summing the values of an array.

REFERENCES

[Anderson88]

Anderson, J. D., Perkins J.A., "Experience Using an Automated Metrics Framework in the Review of Ada Source for WIS", Six Annual Conference on Ada Technology, March 1988, pp. 32-41.

[Barnes89]

Barnes, J. G. P., Programming in Ada, Addison-Wesley Publishing Company, 1989.

[Bell76]

Bell, D. E., LaPadula, L. J., "Secure Computer System: Unified Exposition and Multics Interpretation", Technical Report MTR-2997, Mitre Corporation, Bedford, Ma., March 1976.

[Booch83]

Booch, Grady, Software Engineering in Ada, Benjamin/Cummings Publishing Company, Inc. Reading, Ma, 1983.

[Lamport82]

Lamport, L., Shostak, R., Pease, M., "The Byzantine Generals' Problem", ACM Transactions on Programming Languages and Systems, Vol 4, No. 3, July 1982, pp. 382-401.

[Levine90]

Levine, S., Anderson, J. D., Perkins J.A., "Experience Using Automated Metric Frameworks in the Review of Ada Source for AFATDS", Eight Annual Conference on Ada Technology, March 1990, pp. 597-612.

[Perkins86]

Perkins J. A., Lease, D. M., Keller, S. E., "Experience Collecting and Analyzing Automated Software Quality Metrics for Ada", Fourth Annual National Conference on Ada Technology, March 1986, pp. 67-74.

[Perkins87]

Perkins J. A., Gorzela, R. S., "Experience Using an Automated Framework to Improve the Quality of Ada Software", Fifth Annual National Conference on Ada Technology, March 1987, pp. 277-284.

[Perkins89]

Perkins J. A., "Programming Practices: Analysis of Ada Source Developed for the Air Force, Army, and Navy", TRI-Ada '89, October 1989, pp. 342-354.

[Perkins91]

Perkins J. A., "Programming Practices Relating to Minimal Visibility and Minimal Interaction", Ninth Annual National Conference on Ada Technology, March 1991.

[SPC89]

Software Productivity Consortium, "Ada Quality and Style", Van Nostrand Reinhold, New York, NY.

ABOUT THE AUTHOR

John Perkins is a member of the Software Research Development Group at Dynamics Research Corporation. He has a Bachelor of Science degree in Mathematics from Purdue University and a Master of Science degree in Mathematics from the University of Illinois. He has been involved in the development of a math library and communication software (Digital Message Device), translators for multi-processor scientific computers (Burroughs Scientific Processor and Flow Model Processor), an attribute grammar-based translator-writing system (SSAGS) and static analyzers for assessing the quality of Ada source (AdaMAT). He is currently involved in defining trust metrics for Ada.

PROGRAMMING PRACTICES RELATING TO MINIMAL VISIBILITY AND MINIMAL INTERACTION

J. A. Perkins
Dynamics Research Corporation
Systems Division
60 Frontage Road Andover, MA 01810

ABSTRACT

In this paper, we discuss applying programming practices in Ada that limit the computational model available to the implementor of a given package, subprogram, or task. The goal of the specifiers of a critical segment of Ada source is to limit the implementor's view to the point that 1) only the minimal set of operators and objects are available for use (minimal visibility), and 2) each available operator can operate on a minimal set of these visible objects (minimal interaction), while still providing the ability to implement an efficient version of the desired functionality.

In our research, we examined the relationship of these practices to 1) specific features of the Ada language, 2) the goal of minimal visibility and minimal interaction, and 3) measurements of adherence to this goal. We applied these techniques to the implementation of a stack package.

KEYWORDS

Software practices, security, metrics, Ada

1. INTRODUCTION

Although extensive research has been performed concerning the benefits of code reviews, static and dynamic analysis of code, code testing, formal proofs, and creating "safe subsets" of the Ada language [Cohen89], far less research has been directed at determining how best to use and the benefits of using, the full power of the Ada language to create the minimal computational model required by an implementor of a critical segment of Ada software.

The concept of restricting the computational model through use of guidelines, instead of by removing Ada language features, has been suggested [Preston89]. Additionally, other researchers have proposed source-level programming techniques to limit visibility of data items to only those programs units with both a justifiable need and proper privileges [Keller89].

We investigated 1) applying the programming practices proposed in [Preston89], and 2) combining these practices with other moreunconventional programming practices on a stackpackage to determine how well the Ada language supports the goal of minimal visibility and minimal interaction.

The Ada language supports restricting the computational model available to both consumers and producers of a critical code segment, by providing features such as

separate specifications and bodies for library units,

separate compilation units for nested subunits,

visibility through library access,

private and limited private type declarations,

derived type declarations,

specifications for the mode of parameters,

dynamic constants, and

masking homographs.

Proper use of these features can severely limit the consumers' and producers' ability to unintentionally or maliciously provide unneeded or undesired operational capabilities. In Ada, minimal visibility is less difficult to ensure than is minimal interaction.

In our research, we view an implementor's failure to use a provided operator (object) or interaction between objects as an indication that the implementor 1) has undesired access to an unneeded operator (object), 2) has undesired access to an unneeded potential interaction, or 3) has not completed the coding of the desired functionality. We assume that access to the Ada library (source and object) is tightly controlled, such that no library unit can be accessed, on a compilation unit by compilation unit basis, by a consumer or a producer without permission.

Our discussions focus on 1) programming techniques that restrict the computational model available to consumers and producers of stacks [Section 2], and 2) metrics that measure adherence to these techniques and the goal of minimal visibility and minimal interaction [Section 3].

2. PRACTICES THAT LIMIT MODEL

Booch's implementation of the stack packages uses numerous programming practices that affect both operational and non-operational aspects of software quality [Booch87]. The computational models available to 1) consumers of the specification of stack package and 2) producers of the body of stack package are restricted, by his use of features such as limited private types, programmer-defined exceptions, generic parameters, and parameter modes.

The computational model available to consumers and producers of the stack package can be further restricted by the following practices:

Removing in the declaration of the subprograms the stack as a parameter in those instances where there is a need for only one stack [Section 2.1].

Creating a separate library unit for each item declared in the specification of the stack package [Section 2.2].

Declaring in inner packages the subprograms and exceptions declared in the specification of the *stack package [Section 2.3]*.

Isolating generic parameters in those instances where that parameter is referenced by only one (a few) of the subprograms in the specification of the stack package [Section 2.4].

Using homographs to mask visible items that are not needed by inner structures or subunits [Section 2.5].

Declaring and implementing subprograms that abstract the basic operators for use by the producers of the bodies of the subprograms declared in the specification of stack package [Section 2.6].

Declaring derived types for use by the producers of bodies of those subprograms specified to have two or more parameters of the same type [Section 2.7].

The subsections to follow illustrate how an increased emphasis on minimal visibility and minimal interaction affects 1) the choice of what source-level programming practices are applied, and 2) the ease of validating the functional exactness of the software.

2.1 REMOVE THE OBJECT PARAMETER

Each of Booch's implementations of the package STACK_PACKAGE adheres to the programming practice, "Global information to subprograms should be passed as parameters", by specifying the stack as a parameter of each of the subprograms declared in the package specification. Allowing for slight variations, Booch's specification for each package STACK_PACKAGE is as follows:

```
generic
  type item_type is private;
  type depth_type is (<>);
  -- DEPTH_TYPE is only accessed
  -- by DEPTH_OF.
  --...
package stack_package is
  type stack_type is limited private;
  -- STACK_TYPE is declared in the
  -- specification of STACK_PACKAGE.

  -- The specifications of subprograms
  -- involving actions on a single stack.
  function top_of
    (the_stack_i: in stack_type)
  return item_type;

  function depth_of
    (the_stack_i: in stack_type)
  return depth_type;

  procedure push
    (the_item_i: in item_type;
     the_stack_io: in out stack_type);

  -- ...

  -- The specifications of subprograms
  -- involving actions on multiple stacks.
  function is_equal
    (the_left_stack_i: in stack_type;
     the right_stack_i: in stack_type)
  return boolean;

  procedure copy
    (from_the_stack_i: in stack_type;
     to_the_stack_io: in out stack_type);

private -- stack_package
  -- ...
end stack_package;
```

This parameterization allows the subprograms of any instantiation of package STACK_PACKAGE to be used for any stack declared of the corresponding type STACK_TYPE. Thus, these subprograms are useful when there is a need for only one stack or when there is a need for several stacks of a given type STACK_TYPE. The decision of how many stacks, and the actual declarations of those stacks, is made external to package STACK_PACKAGE. Booch's belief in the benefits of parameterization of the primary object is apparent in his implementations of stacks, queues, lists, trees, and graphs.

Although Booch offers many different forms of stacks, the subprograms of each of these stack packages passes the stack as a parameter. Each form of these packages contains the subprograms IS_EQUAL and COPY. Booch's stack packages support the case where only a single stack is needed, but provide no special support for such a case. Clearly, from a purely functional point of view, no special support for the single stack case is required.

Booch never adheres to the practice, "If there is a need to declare only one object of a given type, then the type for the object and the object should be declared in the body of the package (having the specification that contains the specification of the subprograms acting on that object)". Using this practice, the specification and body for package STACK_PACKAGE is as follows:

```
generic
  type item type is private;
  type depth_type is (<>);
  -- DEPTH_TYPE is only accessed
  -- by DEPTH_OF.
  --...
package stack_package is
  -- The declaration of STACK_TYPE
  -- is moved to the body of STACK_PACKAGE.

  -- The specifications of subprograms
  -- involving actions on a single stack
  -- no longer have a parameter for the
  -- stack.
  function top_of
  return item_type;

  function depth_of
  return depth_type;

  procedure push
    (the_item_i: in item_type);

  -- ...

  -- The specifications of subprograms
  -- involving actions on multiple stacks
  -- are no longer needed.
private -- stack_package
  -- ...
end stack_package;
```

```
package body stack_package is
  type stack_type is ... ;
  -- STACK_TYPE is declared in
  -- the body of STACK_PACKAGE,
  -- ensuring that consumers cannot
  -- declare stacks.
  stack_v: stack_type := ... ;
  -- STACK_V is declared in
  -- the body of STACK_PACKAGE.
  -- This is the only declaration
  -- of a stack.

  -- The bodies of subprograms involving
  -- actions on a single stack are
  -- implemented as before, except the
  -- object representing the stack is no
  -- longer a parameter.

  -- ...

  -- The bodies of subprograms involving
  -- actions on a multiple stacks are no
  -- longer needed.
end stack_package;
```

Adherence to this practice of declaring the single occurrence of the primary object in the body of the package aids in ensuring that 1) only a single stack is declared, and 2) all uses of the subprograms specified in package STACK_PACKAGE result in actions on this single stack, since the stack is declared by a producer of the body of package STACK_PACKAGE inside the body of this package, instead of being declared external to package STACK_PACKAGE by a consumer of the package.

Moreover, adherence to this practice eliminates 1) the need for access of package STACK_PACKAGE by consumers who merely pass the stack as a parameter to other consumers, but do not directly call any of the subprograms specified in this package, and 2) the possibility that a consumer unintentionally or maliciously declares another stack duringmaintenance after the initial validation of the software.

From the perspective of limiting the computational model of consumers of the package STACK_PACKAGE, the impact of this practice is positive. However, the impact on producers of the body of package STACK_PACKAGE is negative, since producers of the subprogram bodies are provided an expanded computational model. The modes used in specifying the stack as a parameter of the subprograms of package STACK_PACKAGE dictate whether the producers of the various subprograms have read-only or read-write access to the stack. Declaring the stack in the body of package STACK_PACKAGE provides read-write access to the producers of each of these subprograms. For instance, the producer of the function TOP_OF is provided read-only access to the stack when the stack is a parameter and is provided read-write access to the stack when the stack is declared as a variable in the body.

2.2 SEPARATE LIBRARY UNITS

Each of Booch's implementations of a stack package provides all the specified operations on the stack for any consumer of an instantiation of these packages. The consumer of the stack requiring only the ability to read the top element on the stack using a call to function TOP_OF is provided other read capability (functions) such as IS_EMPTY, DEPTH_OF, and IS_EQUAL, and write capability (procedures) such as CLEAR, PUSH, POP, and COPY.

The importance of ensuring a read-only view of information when issues of security or safety are involved is emphasized in the research of the Byzantine generals' problem [Lamport82], security models [Bell76], and conceptual invariance [Perkins91]. Placing each of the items declared in the specification of package STACK_PACKAGE in a separate library unit after instantiation allows each consumer of the stack to limit access to only those types, subprograms, or exceptions that are required by that consumer. The consumer requiring only function TOP_OF accesses the library unit containing just this function instead of accessing the more general library unit. Using the programming package, "Create a separate library unit for each item declared in a package specification", results in the following packages for each of the items declared in the specification of package STACK_PACKAGE.

```
with secure_stack_package;
-- SECURE_STACK_PACKAGE is an
-- instantiation of STACK_PACKAGE.
package stack_type_package is

  subtype stack_type is
    secure_stack_package.stack_type;

end stack_type_package;

with secure_stack_package;
-- SECURE_STACK_PACKAGE is an
-- instantiation of STACK_PACKAGE.
with item_type_package;
-- ITEM_TYPE_PACKAGE contained
-- the type used in the
-- instantiation of STACK_PACKAGE.
package top_of_package is

  function top_of
    (the_stack_i: in
       secure_stack_package.stack_type)
  return item_type_package.item_type
  renames secure_stack_package.top_of;

end top_of_package;

-- A separate package for each item
-- declared in STACK_PACKAGE.
-- ...
```

Adherence to this practice of creating separate library units aids in providing control at the level of the library units of the consumers' ability to read and/or write the stack. Consumers not requiring write(read) access to the stack can be denied access to the individual library units that provide such access.

This practice is beneficial in limiting the computational model of consumers of items originally specified in package STACK_PACKAGE. It has no impact on limiting the computational model of the producers of the body of package STACK_PACKAGE.

An important special case of the above programming practice is the practice, "Create a separate library unit containing only a unconstrained subtype of any type declared in the non-private portion of a package specification". In Ada, visibility to the parent type provides capabilities that are not available when only a subtype of that type is visible. For example, visibility to an enumeration type provides access to the corresponding literals whereas visibility to only a subtype of this type provides no access to these literals.

For an in-depth discussion of creating multiple views for objects, see [Keller89].

2.3 NON-DERIVABLE VIEWS

Each of Booch's implementations of the stack packages adheres to the programming practice, "Declare the subprograms for an object in the immediate scope of the type for that object". This practice causes derived types of the parent type to derive the associated programmer-defined subprograms. The programming practice, "Nest the declaration of each subprogram associated with a type in an inner package (in the immediate scope of the type)" prevents derived types from deriving the associated programmer-defined subprograms. Using this practice for preventing derivable subprograms results in the following specification of package STACK_PACKAGE:

```
generic
  type item_type is private;
  -- ...
package stack_package is

  type stack_type is limited private;
  -- STACK_TYPE is declared outside
  -- the inner packages.

  -- The specifications of all the subprograms
  -- involving actions on stack(s) are
  -- encapsulated in inner packages.
  package top_of_package is

    function top_of
      (the_stack_i: in stack_type)
    return item_type;

  end top_of_package;

  -- ...

  package push_package

    procedure push
      (the_item_i: in item_type;
       the_stack_io: in out stack_type);

  end push_package;

  -- ...

private -- stack_package
  -- ...
end stack_package;
```

Assuming that the library unit STACK_TYPE_PACKAGE is created as in the example in Section 2.2, adherence to this practice of creating inner packages prevents a consumer of package STACK_TYPE_PACKAGE from creating a complete view of the package SECURE_STACK_PACKAGE by merely declaring a derived type of the subtype STACK_TYPE.

This practice is beneficial in limiting the computational model of consumers of package STACK_TYPE_PACKAGE. By itself, it has no impact on limiting the computational model of the producers of the body of package STACK_PACKAGE. In Section 2.5, homographs of unneeded items are declared in the body of each of these inner packages to limit the computational model of the producer of the corresponding subprogram.

2. 4 ISOLATE GENERIC PARAMETERS

In Booch's implementation of the unbounded stack package, the function DEPTH_OF is the only subprogram requiring access to the type DEPTH_TYPE.

Actually, the presence of the generic parameter DEPTH_TYPE illustrated in Section 2.1 is a slight variation from Booch's specification. The type NATURAL of package STANDARD is used to represent the depth of the stack in his specification of package STACK_PACKAGE. For the advantages of DEPTH_TYPE as a parameter, see [Perkins88].

When using the programming practice, "If a generic parameter is only accessed by a single subprogram in a generic package, then declare the subprogram *to be a generic subprogram and specify this* parameter as the generic parameter of this new generic subprogram", the specification for package STACK_PACKAGE is as follows:

```
generic
  type item_type is private;
  -- Move DEPTH_TYPE to
  -- declaration of DEPTH_OF.
  --...
package stack_package is
  type stack_type is limited private;
  -- STACK_TYPE is declared in the
  -- specification of STACK_PACKAGE.

  -- The specifications of subprograms
  -- involving actions on a single stack.
  function top_of
    (the_stack_i: in stack_type)
  return item_type;

  -- Declare DEPTH_OF as a
  -- generic function.
  generic
    type depth_type is (<>);
    -- DEPTH_TYPE is declared as
    -- parameter of DEPTH_OF.
  function depth_of
    (the_stack_i: in stack_type)
  return depth_type;

  procedure push
    (the_item_i: in item_type;
     the_stack_io: in out stack_type);

  -- ...

  -- The specifications of subprograms
  -- involving actions on multiple stacks.
  function is_equal
    (the_left_stack_i: in stack_type;
     the right_stack_i: in stack_type)
  return boolean;

  procedure copy
    (from_the_stack_i: in stack_type;
     to_the_stack_io: in out stack_type);

private -- stack_package
  -- ...
end stack_package;
```

Adherence to this practice of isolating generic parameters eliminates visibility to type DEPTH_TYPE from the producers of the other subprograms encapsulated in package STACK_PACKAGE. Granted, the information concerning the depth of the stack is not likely to be sensitive. On the other hand, the generic parameter ITEM_TYPE could easily represent secure information, and not all the subprograms encapsulated in package STACK_PACKAGE require access to this type. However, the use of type ITEM_TYPE in the non-private declaration of type NODE_TYPE causes difficulty in isolating this parameter. See [Keller89] for a discussion of techniques for isolating ITEM_TYPE when package STACK_PACKAGE is not implemented as a generic package.

From the perspective of limiting the computational model of producers of the other subprograms implemented in the body of package STACK_PACKAGE, the impact of this practice is positive. However, the impact on the consumers of the package is negative, since consumers are provided an expanded computational model. The consumer of an instantiation of package STACK_PACKAGE can instantiate multiple DEPTH_OF functions. This expansion in the computational model for consumers of the stack can be eliminated by creating, as recommended in Section 2.2, a separate library unit for each of the subprograms in the instantiation of package STACK_PACKAGE, where the separate library unit corresponding to the generic function DEPTH_OF is an instantiation of DEPTH_OF.

2.5 MASKING HOMOGRAPHS

In Booch's implementation of function DEPTH_OF and procedure COPY of the unbounded version of package STACK_PACKAGE, the input parameters are not referenced inside the LOOP statements that control the transversal from one element of the stack to the next. In both of these subprograms, local variables of type STACK_TYPE are declared to support these transversals of the stacks. Booch's implementation of procedure COPY is as follows:

```
procedure copy
  (from_the_stack_i: in stack_type;
   to_the_stack_io: in out stack_type)
is
  -- COPY does not call itself recursively.
begin -- copy
  -- null_from_the_stack_if:
  if from_the_stack_i = null
  then
    to_the_stack_io := null;
  end if; -- null_from_the_stack_if
  copy_block:
  declare
    from_stack_lv: stack_type
      := from_the_stack_i;
    -- Last reference to FROM_THE_STACK_I.
    to_stack_lv: stack_type :=
      new node_type'
        (the_item =>
           from_stack_lv.the_item,
         next => null);
  begin -- copy_block
    to_the_stack_io := to_stack_lv;
    -- Last reference to TO_THE_STACK_IO.
    from_stack_lv := from_stack_lv.next;
    -- There are no references to
    -- FROM_THE_STACK_I and TO_THE_STACK_IO
    -- inside COPY_LOOP.
    copy_loop:
    loop
      -- from_stack_lv_null_if:
      if from_stack_lv = null
      then
        return;
      end if; -- from_stack_lv_null_if
      to_stack_lv.next :=
        new node_type'
         (the_item =>
            from_stack_lv.the_item,
          next => null);
      to_stack_lv :=  to_stack_lv.next;
      from_stack_lv := from_stack_lv.next;
    end copy_loop;
  end copy_block;
exception -- copy
when storage_error =>
  raise overflow;
end copy;
```

In Ada, homographs can be declared in an inner structure to prevent direct references to items required only by an outer structure. Using the practice, "Declare a homograph of a visible item if direct references to the name of that item are not required", results in the following implementation of procedure COPY:

```
separate (stack_package.copy_package)
procedure copy
  (from_the_stack_i: in stack_type;
   to_the_stack_io: in out stack_type)
is
  copy: constant := 0;
  -- Mask COPY.
  -- Declare homograph of the
  -- PROCEDURE COPY to indicate
  -- that this procedure does not
  -- call itself recursively.
begin -- copy
  -- null_from_the_stack_if:
  if from_the_stack_i = null
  then
    to_the_stack_io := null;
  end if; -- null_from_the_stack_if
  copy_block:
  declare
    storage_error,
    overflow: constant := 0;
    -- Mask STORAGE_ERROR, and OVERFLOW.
    from_stack_lv: stack_type
      := from_the_stack_i;
    -- Last reference to FROM_THE_STACK_I.
    from_the_stack_i: constant := 0;
    -- Mask FROM_THE_STACK_I.
    -- Declare a homograph of
    -- parameter FROM_THE_STACK_I
    -- to indicate visibility to the
    -- parameter is no longer needed
    to_stack_lv: stack_type :=
      new node_type'
        (the_item =>
           from_stack_lv.the_item,
         next => null);
    stack_type: constant := 0;
    -- Mask STACK_TYPE.

  begin -- copy_block
    to_the_stack_io := to_stack_lv;
    -- Last reference to TO_THE_STACK_IO.
    mask_to_the_stack_io_block:
    declare
      to_the_stack_io: constant := 0;
      -- Mask FROM_THE_STACK_I.
      -- Declare a homograph of
      -- parameter FROM_THE_STACK_I
      -- to indicate visibility to the
      -- parameter is no longer needed
    begin -- mask_to_the_stack_io_block
      from_stack_lv := from_stack_lv.next;
      -- There are no references to
      -- FROM_THE_STACK_I and TO_THE_STACK_IO
      -- inside COPY_LOOP.
      copy_loop:
      loop
        -- from_stack_lv_null_if:
        if from_stack_lv = null
        then
          return;
        end if; -- from_stack_lv_null_if
        to_stack_lv.next :=
         new node_type'
         (the_item => from_stack_lv.the_item,
          next => null);
        to_stack_lv :=  to_stack_lv.next;
        from_stack_lv := from_stack_lv.next;
      end copy_loop;
    end mask_to_the_stack_io_block;
  end copy_block;
exception -- copy
when storage_error =>
  raise overflow;
end copy;
```

Adherence to this practice of declaring masking homographs aids in ensuring that successful compilation of procedure COPY implies that1) procedure COPY is never called inside COPY, and2) parameters FROM_THE_STACK_I and TO_THE_STACK_IOare never referenced inside the loop COPY_LOOP.

The homographs of the exceptions OVERFLOW andSTORAGE_ERROR and the type STACK_TYPE are declaredto limit direct references to these items in the inner most block and LOOP of procedure COPY.

In the above case, the producer of the body of procedure COPY is using homographs to limit the computational model available inside inner regions of procedure COPY. In the example below, the producers of the body of package STACK_PACKAGE and using homographs to mask items not needed by theproducer of the body of procedureSTACK_PACKAGE.COPY_PACKAGE. COPY.

```
package body stack_package is
  -- Mask everything in package
  -- STANDARD except
  --   BOOLEAN,
  --   CONSTRAINT_ERROR, and
  --   STORAGE_ERROR.
  -- Declare homographs to hide
  -- unneeded items in STANDARD
  -- from the producers of the
  -- inner packages specified in
  -- STACK_PACKAGE.
  standard,
  integer,
  natural,
  positive,
  character,
  -- ...
  task_error,
  stack_package: constant := 0;

  package top_of_package is separate;
  -- ...
  package copy_package is separate;
end stack_package
separate (stack_package)
package body copy_package is

  -- Mask everything visible
  -- to COPY_PACKAGE except
  --   COPY,
  --   STACK_TYPE,
  --   NODE_TYPE,
  --   STORAGE_ERROR, and
  --   OVERFLOW
  -- Declare masking homographs
  -- for visible items not
  -- needed by COPY.
  boolean,      -- standard_package
  constraint_error,
  depth_type, -- generic parameters
  item_type,
  top_of,       -- subprograms
  is_empty,
  depth_of,
  is_equal,
  push,
  pop,
  clear,
  underflow,   -- exceptions
  copy_package: constant := 0;
```

```
      -- The computational model available
      -- to the producer of the body of
      -- procedure copy has been restricted
      -- by the producer of the package
      -- specification for package STACK_PACKAGE,
      -- by the producer of the package
      -- body for package STACK_PACKAGE, and
      -- by the producer of the package
      -- body for package COPY_PACKAGE.
      procedure copy
        (from_the_stack_i: in stack_type;
         to_the_stack_io: in out stack_type)
      is separate;

    end copy_package;
```

Adherence to this practice of declaring masking homographs aids in ensuring that successful compilation of procedure COPY implies 1) no references to unneeded items from package STANDARD, 2) no references to the other subprograms declared in package STACK_PACKAGE, and 3) no references to the generic parameters of package STACK_PACKAGE. The homographs of packages STANDARD, STACK_PACKAGE, and COPY_PACKAGE are declared to prevent qualified references to the items declared in these packages. Adherence to this practice greatly increases the likelihood that an unintentional reference to an unneeded item is detected at compile time.

The declaration of these masking homographs has no effect on the consumers of package STACK_PACKAGE.

The use of the universal integer type and 0 in declaring homographs is merely a matter of convenience. When non-numeric objects are involved, this appears to be a reasonable choice. When numeric objects are involved, type BOOLEAN and false can be used. In our library, we have, solely for the purpose of declaring these masking homographs, package HIDE_PACKAGE containing the private type HIDE_TYPE and the constant HIDE. We always declare masking homographs as constants.

Creating masking homographs for operators such as "+" and "." is more difficult; however, such homographs are valuable in limiting the run time model of the producers. The homographs for such operators are usually implemented to raise a run time exception to indicate that the call to the operator is unintentional or unexpected.

Combining the examples above, the computational model available to the producer of the body of procedure COPY is limited by the producers of 1) the specification of package STACK_PACKAGE, 2) the body of package STACK_PACKAGE, 3) the body of package COPY_PACKAGE, and 4) the body of procedure COPY.

2.6 ABSTRACT BASIC OPERATORS

In Booch's implementation of the unbounded version of the package STACK_PACKAGE, the producers of the bodies of the subprograms are provided direct access to the underlying data structures for types STACK_TYPE and NODE_TYPE. Using the programming practice, "Provide subprograms (in the body of the encapsulating package) which duplicate the basic operators of each non-private type", allows the following implementation of the body of procedure COPY:

```
procedure copy
  (from_the_stack_i: in stack_type;
   to_the_stack_io: in out stack_type)
is
begin -- copy
  -- null_from_the_stack_if:
  if from_the_stack_i = empty_stack
  -- Constant EMPTY_STACK replaces null.
  then
    to_the_stack_io := empty_stack;
    -- Constant EMPTY_STACK replaces null.
  end if; -- null_from_the_stack_if
  copy_block:
  declare
    from_stack_lv: stack_type
      := from_the_stack_i;
    to_stack_lv: stack_type :=
      make(the_item(from_stack_lv),
           empty_stack);
    -- Function MAKE replaces NEW.
    -- Function THE_ITEM replaces .the_item.
    -- Constant EMPTY_STACK replaces null.
  begin -- copy_block
    to_the_stack_io := to_stack_lv;
    from_stack_lv := next(from_stack_lv);
    copy_loop:
    loop
      -- from_stack_lv_null_if:
      if from_stack_lv = empty_stack
      -- Constant EMPTY_STACK replaces null.
      then
        return;
      end if; -- from_stack_lv_null_if
      next(to_stack_lv,
           make(the_item(from_stack_lv),
                empty_stack));
      -- Procedure NEXT replaces .next.
      -- Function MAKE replaces NEW.
      -- Function THE_ITEM replaces .the_item.
      -- Constant EMPTY_STACK replaces null.
      to_stack_lv :=  next(to_stack_lv);
      -- Function NEXT replaces .next.
      from_stack_lv := next(from_stack_lv);
      -- Function NEXT replaces .next.
    end copy_loop;
  end copy_block;
-- The exception STORAGE_ERROR
-- is handled by function MAKE.
end copy;
```

Adherence to this practice of abstracting basic operators eliminates the producer's need to rely on the non-private type STACK_TYPE view of the stack object. The producer of the body of procedure COPY relies instead on a private view of type STACK_TYPE, the constant EMPTY_STACK, the functions MAKE, THE_ITEM, and NEXT, and the procedure NEXT, which are provided by the producer of the body of package STACK_PACKAGE. Compare the above implementation of procedure COPY to the implementation in Section 2.5.

The declaration of these subprograms for the basic operators has no effect on the consumers' view of package STACK_PACKAGE. Of course, creation of these subprograms has a negative impact on both code space and run time performance.

2.7 DERIVED TYPES

In Booch's implementation of the bodies of function IS_EQUAL and procedure COPY, type STACK_TYPE is used in the declaration of both stack parameters. Using the programming practice, "Declare (in the body of the encapsulating package) derived types of the parent type and provide the necessary operators with the appropriate functions signatures, whenever multiple parameters for a specified subprogram (in the specification of the encapsulating package) are of the same type", the implementation of the body of procedure COPY is as follows:

```
separate(stack_package.copy_package)
procedure copy
  (from_the_stack_i: in from_stack_type;
   -- FROM_STACK_TYPE replaces STACK_TYPE.
   to_the_stack_io: in out to_stack_type)
   -- TO_STACK_TYPE replaces STACK_TYPE.
is
begin -- copy
  -- null_from_the_stack_if:
  if from_the_stack_i = from_empty_stack
  -- FROM_EMPTY_STACK replaces EMPTY_STACK.
  then
    to_the_stack_io := to_empty_stack;
    -- TO_EMPTY_STACK replaces EMPTY_STACK.
  end if; -- null_from_the_stack_if
  copy_block:
  declare
    from_stack_lv: from_stack_type
      := from_the_stack_i;
    -- FROM_STACK_TYPE replaces STACK_TYPE.
    to_stack_lv: to_stack_type :=
      to_make(from_the_item(from_stack_lv),
              to_empty_stack);
    -- TO_STACK_TYPE replaces STACK_TYPE.
    -- TO_MAKE replaces MAKE.
    -- FROM_THE_ITEM replaces THE_ITEM.
    -- TO_EMPTY_STACK replaces EMPTY_STACK.
  begin -- copy_block
    to_the_stack_io := to_stack_lv;
    from_stack_lv := from_next
                       (from_stack_lv);
    copy_loop:
    loop
      -- from_stack_lv_null_if:
      if from_stack_lv = from_empty_stack
      -- FROM_EMPTY_STACK replaces
      -- EMPTY_STACK.
      then
        return;
      end if; -- from_stack_lv_null_if
      to_next(to_stack_lv,
              to_make(from_the_item
                        (from_stack_lv),
                      to_empty_stack);
      -- TO_NEXT replaces NEXT.
      -- TO_MAKE replaces MAKE.
      -- FROM_THE_ITEM replaces THE_ITEM.
      -- TO_EMPTY_STACK replaces EMPTY_STACK.
      to_stack_lv :=  to_next(to_stack_lv);
      -- TO_NEXT replaces NEXT.
      from_stack_lv := from_next
                         (from_stack_lv);
      -- FROM_NEXT replaces NEXT.
    end copy_loop;
  end copy_block;
-- The exception STORAGE_ERROR
-- is handled by function TO_MAKE.
end copy;
```

The producer of the body of package COPY_PACKAGE 1) declares type FROM_STACK_TYPE and TO_STACK_TYPE as derived types of type STACK_TYPE, and 2) specifies and implements (using the abstracted basic operators of type STACK_TYPE) the subprograms FROM_EMPTY_STACK, FROM_NEXT, FROM_THE_ITEM, TO_EMPTY_STACK, TO_MAKE, TO_NEXT (function), and TO_NEXT (procedure). The example below outlines this implementation of package COPY_PACKAGE.

```
separate (stack_package)
package copy_package is

  type from_stack_type is new stack_type;
  type to_stack_type is new stack_type;

  function from_empty_stack
  return from_stack_type;

  function from_next
    (from_stack_i: in from_stack_type)
  return from_stack_type;

  function to_empty_stack
  return to_stack_type;

  function from_the_item
    (from_stack_i: from_stack_type)
  return item_type;

  function to_make
    (to_item_i: in item_type;
     to_stack_i: in to_stack_type)
  return to_stack_type;

  function to_next
    (to_stack_i: in to_stack_type)
  return to_stack_type;

    procedure to_next
      (to_stack_io: in out to_stack_type;
       to_stack_i: in to_stack_type);
    -- ...

    procedure copy
      (from_the_stack_i: in from_stack_type;
       to_the_stack_io: in out to_stack_type)
    is separate;

    -- The version of COPY having two
    -- parameters of the same type
    -- is implemented by calling the
    -- version of copy having the
    -- two parameters of different
    -- types.

  end copy_package;
```

Adherence to this practice aids in ensuring that the producer of the body of procedure COPY does *not unintentionally or maliciously perform an* action on the wrong stack. The function signature for the function FROM_THE_STACK ensures that items are read from the object pointed to by the variable FROM_STACK_LV and not from the object pointed to by the variable TO_THE_STACK_LV. The function signatures for the other subprograms ensure that the transversals of the two stacks are not confused.

The declaration of these derived types has no effect on the consumers' view of package STACK_PACKAGE.

In Hilfinger's investigation of Ada, he recommended the removal of derived types from the language [Hilfinger81]. Our research indicates that derived types are a valuable mechanism for limiting the interaction between multiple objects of identical structure. In our examination of a tree package supplied by the Air Force [BSTS89], we illustrated that important properties concerning the integrity of the tree can be ensured by declaring derived types of the tree type for the parent node and child node of the tree during grafting (insertion). Grafting requires different fields to be updated in the parent node than in the child node. The declaring of derived types prevents the updating by the child of a field required by the parent but not required by the child and vice versa. For instance, the number of children can be updated in the parent node but not in the child node. For more details concerning this implementation of the tree package, see [SDI90].

3. METRICS FOR VISIBILITY AND INTERACTION

The degree to which the software meets the goal of minimal visibility and minimal interaction can be measured 1) indirectly, by measuring the proportion of times the proposed programming practices are used when use of such a practice is possible (e.g. What proportion of the package specifications containing a declaration of a type and operators on that type contain the declaration of these operators in inner packages); or 2) directly, by measuring a) minimal visibility, in terms of the proportion of items that are visible to a unit that are referenced by that unit and b) minimal interaction, in terms of the proportion of non-keywords in the source that are not able to be replaced, one occurrence at a time, by another non-keyword without causing the library unit to become uncompilable.

Both forms of metrics are important, since the indirect measures have related actions that improve the scores for the direct measures, and the direct measures indicate how close to minimal is the computational model provided to the producer of a unit.

4. CONCLUSION

Although, for most applications, is it impossible to completely achieve minimal visibility and minimal interaction using the Ada language, our research indicates that there exists practical programming techniques for greatly reducing the computational model available to producers and consumers of critical segments of Ada source. More importantly, our research indicates that important properties relating to exactness can be ensured by limiting both visibility and interaction at the level of the Ada source . Often, the properties ensured by these techniques are difficult to validate during the testing and/or operation of the software. For more detailed information concerning the practices presented here and other practices that can be used to limit the computational model of implementors, see [SDI90].

Metrics have been defined that measure important aspects of visibility and interaction. Static analysis tools to automate the collection of these metrics directly from the Ada source could be built. We have on going efforts in both of these areas.

ACKNOWLEDGMENT

The research on programming practices discussed in this paper was performed by Dynamics Research Corporation under contract SDIO 84-88-C-0020, which was awarded to GE Aerospace by the Office of the Secretary of Defense Strategic Defense Initiative Organization, Pentagon, Washington D.C.

REFERENCES

[Booch87]

Booch, Grady, Programming in Ada. Benjamin/Cummings Publishing Company, Inc. Reading, MA, 1987.

[BSTS89]

RFP F04701-89-R-008, Part IV, Section L, Annex B, pp. 93-98, USAF/AFSC HQ Space Systems Division, Los Angeles, CA.

[Cohen89]

Cohen, N., Carre, B. A., McHugh, J., Preston, D., Smith, M. K., Van Scoy, R., Wichmann, B., Panel on Safe Subsets of Ada, TRI-Ada 89, October 1989, pp. 244-254.

[Hilfinger81]

Hilfinger, P. N., Abstraction Mechanisms and Language Design, The MIT Press, 1983.

[Keller89]

Keller, S. E., Perkins, J. A., O'Leary, K., "Layering and Multiple Views of Data Abstraction in Ada: Techniques and Experiences", TRI-Ada 89, October 1989, pp. 342-354.

[Perkins88]

Perkins J. A., "The Myth: Anyone Can Code the Software If the Requirements and Design Are ...", TRI-Ada 88, October 1988, pp. 258-273.

[Perkins91]

Perkins, J. A., "The Mystery: Why Do Many of the Variables Declared in Ada Programs Model Conceptually Invariant Objects?", Ninth Annual National Conference on Ada Technology, March 1991.

[Preston89]

Preston, D., "Panelist Viewpoint: Ada and the Trusted Computer System Evaluation Criteria", TRI-Ada 89, October 1989, pp. 247.

[SDIO90]

Trusted Software Development Methodology Report - Draft 2, Appendix B, Strategic Defense Initiative Organization, Washington, D.C., January, 1990.

AUTHORS' INDEX

Name	Page
Archer, Thomas S.	50
Arden, William	69
Barner, Wanda	32
Bassiouni, M.	16
Berggren, Peter	
Bishop, Michael	206
Blankenship, P	181
Bollinger, Terry B.	255
Brown, David	174
Burnham, C. Alan	87
Chester, Rowena O.	39
Chiu, M.	16
Clark, Peter G.	223
Coleman, Don M.	135
Disbrow, David	249
Elrad, Tzilla	25
Fitzgerald, Jr., Joseph C.	114
Gerardi, Robert D.	87
Gotterbarn, Donald	120
Gray, Lewis	55
Henry, Sallie	107
Hess, Richard	233
Ho, Peter	87
Hooper, James W.	39
Howden, Bill	128
Joiner, II, Harry	87
Kang, C. S.	72
Kitaoka, Beverly	
Kozlowski, Catherine	100
Krous, Ella L.	150
Lawhead, Pamela B.	233
Leach, Ronald J.	135
Levy, Steven H.	92
Li, Wei	107
Lobsitz, Richard M.	223
Lukman, Joseph T.	200
Martin, David	249
Miller, William L.	9
Miriyala, Shakuntala	25
Moore, Freeman L.	170
Mulholland, Sandra L.	188
McGowan, Clem	255
McKay, Amber	152
McKay Charles	206
Newsome, Tina L.	160
Papanicolaou, Basil	45
Park, E. K.	72, 164
Pederson, Richard	152
Perkins, John A.	282, 294
Pfleeger, Shari L.	114
Piper, JoAnne	32
Reid, Thomas F.	243
Rogers, Kathy	206
Royce, Walker E.	174, 181
Rusis, E.	181
Schaefer, Carl	142
Schettino, John	100
Selig, Calvin	107
Skove, F. A.	72
Spegele, USMC, J. J.	164
String, C. Robert	223
Thompson, J.	16
Walker, James E.	238
Wieand, Bruce	128
Willis, B.	181
Zeil, Steven J.	80

Portability and Ada ... Some Suggestions

David E. Johnson

UNIVERSITY OF MISSISSIPPI

Abstract

This paper examines the portability of programs written in Ada by discussing recent experiences in developing a large software package in Ada on an IBM 370 and then porting it to a Sun 3/60. The paper examines potential pitfalls in portability, especially in dealing with I/O issues and word size. Finally, the results of the porting of the software package are examined closely and evaluated critically in a case study environment so that the design problems can be avoided in future projects by every member of the software development team.

1 Introduction

Ada was designed with portability in mind, however, some features of the language may frequently be used in non-portable ways. The most common approach taken in developing a portable application entails extracting the machine-dependent routines and data-types into separate packages which are then used by the machine-independent ones.

A software engineering class at the University of Mississippi recently developed a statistical software package using Ada. Details of its design as well as portability problems encountered during development are discussed below.

2 Overview of STATPLOT

Our project was to develop a statistical library for use in a larger project. This library was to be "with"ed (linked) in by higher-level routines written by other development teams whose projects were not disclosed to us.

The original system requirements specification described a package which was to be run on an IBM MVS-based system. The package was to take an input file and a dataset definition file as input, process the data as needed, and send the output to the screen, a printer, or a file. (see Figure 1)

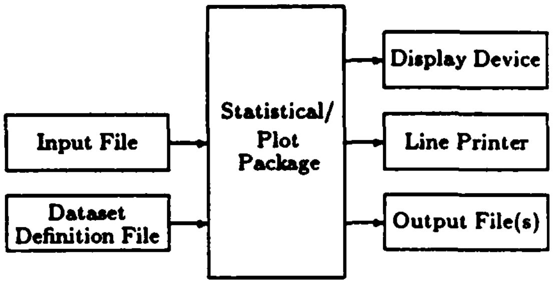


Figure 1: Statistical/Plot Package Interfaces

The MVS dataset input file format is shown in Figure 2. Each dataset contains all possible fields of data on which processing is possible. The `header` is a four character string that identifies the data set. The `length` is the length of the data set (including the `header` and `length` fields). The `data` field contains the data for each field in the record whose structure is defined in the **Dataset Definition File**.

Figure 3 shows the format of the dataset definition file (DDF). To retrieve a field from the file, the desired `field ID` is looked up in the DDF to obtain its data `type` and its `start` offset in the input data file `data` field. Then the desired field may be extracted. Note that the record structure as well as the field types and sizes may change at any time. These record formats pose problems in writing Ada programs. Ada is a strong, statically typed language and handling MVS datasets would require use of unchecked conversions and other non-portable features of Ada thus producing an undesirable program.

MVS Data-sets were to be sent to the statistical routines for analysis and then a resulting data-set and/or value would returned. The data-sets contained all fields collected from the source and each statistical routine was responsible for extracting the specified fields.

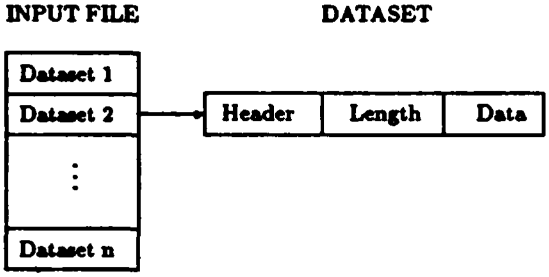


Figure 2: Input File Format

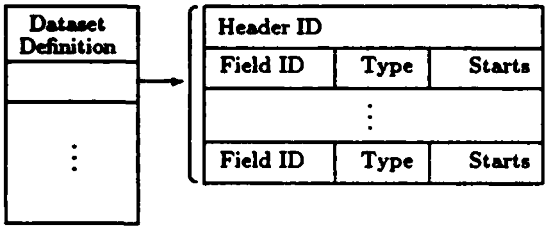


Figure 3: Datset Definition File Format

Our actual development environment was to be IBM CMS-based system but since Ada has features encouraging portable coding, this environment was deemed satisfactory.

The design specifications were divided among development teams who would be responsible for further design refinements and code development for each particular segment of the design. Each team was required to research the appropriate algorithms and choose the best, based on the requirements. After a preliminary research period, all groups met to discuss the overall design.

3 Initial Design Decisions

Early planning focused on how to interface Ada to MVS datasets and how these could be simulated in CMS. It was assumed that in actual use the dataset structure would change infrequently and thus might be feasible to place the data definition statically into the Ada package as a record. However, it was later noted that we could not guarantee that Ada datatypes would match up correctly with the MVS data.

With these problems in mind, as well as the time delay of the interaction with the project coordinators, the I/O routines were separated into an independent Ada package and called "Mystical I/O" since its functionality and interface were unknown. Also, with time constraints placed on us by the system requirements, we could develop our own file formats for internal testing. Then, at delivery time we could provide an untested MVS I/O package for datasets as well as our test setup.

Ada packages allowed our I/O data format to be hidden from the calling procedure as well the the actual user. The other development teams only needed to know the generic interface to the I/O Package. Another benefit of I/O isolation is the data file format may change without affecting the higher level routines.

Since I/O was separated, the other teams concentrated on their algorithms and did not worry about a specific data format. This was extremely nice for the I/O team because contractor requested changes to the data file formats could be done easily.

Output of the statistical routines could be sent to a printer, the screen, or a text file. To isolate system dependency issues surrounding device handling, each statistical routine placed its output in a text file whose name was obtained via a call to the I/O package routine NEW_FILE_NAME. This routine took the name of the calling procedure as a parameter and returned a unique file name consisting of the procedure name and the time of day as an eight digit quantity.

After all output was generated, another I/O routine, SEND_FILE_TO_DEVICE, wass called to send the output file to the appropriate device or leave it in a file. Only this device output routine knew the specifics of interfacing to the appropriate device. Thus, when porting the I/O package to another system, the output handling could be modified without affecting any statistical routines.

4 Further Designs

The production environment for this project was moved from MVS to a UNIX system due to the introduction of the IBM RS/6000 very late in the development process. The MVS data-sets also became obsolete because UNIX does not provide file types such as data-sets, index-sequential, etc. All UNIX files are just a collection of bytes.

To simplify the file interface to the statistical routines, the calling procedures were to pass only data which was necessary for the analysis in a text, readable file. This data would be processed and if necessary, returned in a newly-created text file. Although we continued development under CMS, we used this scheme in hopes that porting the final code to UNIX would be extremely simple.

Due to the file format changes implementation was simplified tremendously. The I/O routines could now return all values found in each record to the caller for processing. However, one problem still existed: values in the data files may be of different types. Each statistical routine needed to be able to deal with calls containing various types. It was realized, though, that by converting all data in the file to LONG_FLOAT's, no data precision would be lost. This also allowed the other routines to deal only with the LONG_FLOAT's which simplified their implementation greatly.

The final I/O design provided functions (see Figure 4) to read and write records (a line of values in the file), open and close files, create file names for temporary files, and maintenance of a error logging file. The read and write record procedures, GET_RECORD_FIELD and PUT_RECORD_FIELD, read/wrote from 1-4 fields depending on the higher level requirements. Each statistical routine "knows" how many fields it is designed to work with (based on the requirements specification) and issues a call to GET_RECORD_FIELD with the number of arguments equal to the number of fields allowed. GET_RECORD_FIELD is overloaded to handle this type of interface and makes the code extremely easy to follow.

The open function would read the file header, record the record format and return the number of fields present to the caller which could check for problems. Figure 5 shows an outline of the algorithm used in each statistical routine to process an input file.

```
Data Files
  CLOSE
  COPY_RECORD
  CREATE
  DELETE
  END_OF_FILE
  GET_CURRENT_RECORD_NUMBER
  GET_RECORD
  GET_RECORD_FIELD
  MERGE_TWO_FILES_TO_ONE
  OPEN
  PUT_RECORD
  PUT_RECORD_FIELD

Log File
  CLOSE_LOG_FILE
  OPEN_LOG_FILE
  PUT_LOG_MESSAGE

Output Files
  SEND_FILE_TO_DEVICE

Miscellaneous
  NEW_FILE_NAME
```

Figure 4: STATPLOT_IO routines

```
STATPLOT_IO.OPEN( ... );

while not STATPLOT_IO.END_OF_FILE loop

  begin                        - - DATA ERROR block
    STATPLOT_IO.GET_RECORD_FIELD( x, y );
  exception
    when STATPLOT_IO.DATA_ERROR =>
      STATPLOT_IO.PUT_LOG_MESSAGE( ... );
  end;                         - - DATA ERROR block

    - - DO PROCESSING - -

end loop;

STATPLOT_IO.CLOSE( STATPLOT_IO.INFILE );
```

Figure 5: Generic STATPLOT algorithm

5 I/O Package Advantages

- **Calling functions were not aware of file structure**

  Each statistical routine dealt only with the abstraction of the input file having one or more fields. The request issued by the routine to read a record conveyed the number of fields of interest. Since the interface is very simple and has no direct relationship with the file structure, the input file structure could be changed without affecting any of the calling procedures.

- **Data returned was of one type**

The actual fields in the input file may be of the following types: INTEGER, NATURAL, POSITIVE, FLOAT, or LONG_FLOAT. However, the values returned by the GET_RECORD_FIELD procedure are always of the type LONG_FLOAT. Therefore, the statistical routines need not be concerned with the data type of the input fields.

- Temporary file names could be formatted by I/O as desired

  When a statistical routine needs to create an output file it requests a unique name from the I/O package. The algorithm used to create this new file name may be changed at any time without affecting the calling procedures.

- Allowed other functions to be written without I/O

  With I/O isolated in a separate package, the statistical functions could be written very simply. No messy file handling code was necessary leaving the bulk of the code to just the actual statistical algorithm.

6 Problem not addressed

- Actual file names could not be made transparent

  Even though file names could be created dynamically, file names embedded in the code would not be portable. The number of static file names could be reduced, but if any exist they will be inherently non-portable.

7 Numeric Quantities

Numeric types are perhaps the most problematic of all portability problems. Sizes and precision of numeric types differ tremendously across implementations of all languages, including Ada. Ada provides very nice facilities to extract limits and sizes from numeric types. However, using an INTEGER type to define a variable may give you 32 bits of storage on one system and 64 bits on another. The name does not accurately describe the size of the storage. Also, restricting yourself to a minimal common denominator, such as 16 bits, may be overly restrictive and not always possible.

It may be useful to define a set of abstract types in an Ada package for use in the system. This way, when porting to another system, the appropriate system types will be mapped onto the local types using Ada attributes such as SIZE, FIRST or LAST. Portability will be increased because only one section of code needs to be changed.

Our contractor tried to do this by providing a "system.spc" package of types for the project. It define SHORT_INTEGER and INTEGER. The INTEGER type was defined to be a 32 bit number because of their implementations. The size of an INTEGER on our IBM 3084 is 64 bits wide. At first this problem may seem minimal, however, consider the following scenario.

> A routine keeps a count of some values. In testing, all combinations of values which were forseen were used. After completing the project the system runs for about 6 months. At that time the software crashes because of a numeric overflow.

What may have happened is that in testing the size of the INTEGER may be 64 bits and the actual system may be using a 32 bit INTEGER. Even though the implementor may have assumed a 64 bit INTEGER, it was not explicitly expressed and the error eventually surfaced.

For integer types it may be more appropriate to create new types with names like: BYTE, INTEGER_16, INTEGER_32, UNSIGNED_32, INTEGER_64, etc. In this way, someone porting the system will map these types onto the system implementation types correctly. Therefore, if a type cannot be correctly mapped, it will be known that other work will need to be done.

We saw this type problem when porting the code from the IBM 3084 to a Sun 3/60. The Sun 3/60 has an INTEGER size of 32 bits. However, while coding NEW_FILE_NAME, the function to create temporary file names, we actually used numbers that may have exceeded a 32 bit quantity. This was not noticed until testing was done on the Sun. An overflow was detected in this routine which caused all others to fail since they all called this one. Since the contractor did not define a LONG_INTEGER and since portability was a factor, the handling of such large numbers was done in two halves: the high-order 4 digits and the low-order 4 digits. Any manipulation done on the entire number was done as a LONG_FLOAT instead.

This solution works in all cases where the LONG_FLOAT could handle these amounts (almost all). However, this solution is a kludge. A long integer could have been defined to solve the problem.

However, we were not allowed to change the definitions. Luckily, though, we were able to catch this problem at the initial testing stages.

Floating point numbers are more complicated. Knowing the size and range of the type is of minimal use [2]. Base representation varies across systems and actual implementations. Precision may be a better mapping quantity. Thus if you require 6 digits of accuracy, a type of FLOAT6 may be appropriate, or for 12 digits maybe FLOAT12. Thus when porting these are mapped appropriately.

8 Suggestions

In conclusion, I make the following suggestions:

Do not focus or worry about I/O until late in detailed design. Work on the I/O package in isolation and let the other teams simulate input as need until the I/O package interface can be standardized. Also, do not standardize the I/O interface until implementation details can be tested.

Do not rely on file name formats. Inside the I/O package, make sure that file name lengths (even filename, filetype) are defined as constants not hardcoded. Also, different system use different separators. CMS uses spaces whereas UNIX uses a period. This should also be a constant. A file name type should be defined and exported for other users.

Do not rely on size of words or numeric types. At the highest level have a small package of defined types which are mapped onto the particular environment using attributes if possible.

References

[1] Olivier Lecarme, Mireille Pellissier Gart, and Mitchell Gart. *Software Portability with Microcomputer Issues.* McGraw-Hill, New York, New York, 1989.

[2] Seth M. Lowrey. On the use of ada in mathematical applications. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles.* ACM, 1991.

[3] John Nissen and Peter Wallis, editors. *Portability and style in Ada.* The Ada Companion Series. Cambridge University Press, Cambridge, 1984.

David E. Johnson received the B.S degree in Computer Science from the University of Mississippi in 1988.

He is a graduate student in Computer Science at the University of Mississippi with interests in operating systems and languages.

David E. Johnson
Dept. of Computer and Information Science
302 Weir Hall
University, MS 38677
(601) 232-7396
dave@cs.olemiss.edu

ADA SIMULATION PROGRAM OF A MULTI-PROCESSOR ENVIRONMENT

Steven K. Iwohara and Dar-Biau Liu

Department of Computer Science and Engineering
California State University, Long Beach
Long Beach, California 90840

Abstract

A simulation, implemented in the Ada programming language, of a dynamic task-scheduling algorithm has been run on a VAX 8530 for the Advanced Multi-Purpose Support Environment's (AMPSE) distributed executive. this is a specially designed network, replicating shared memory, developed to support real-time multiprocessor simulation applications. In this environment, the computers may execute different processes at the same time, or communicate with each other via SMARTNet. The Ada language was chosen for the simulation as it closely models the real world. Its multitasking features allow both synchronous and asynchronous operations, and Ada's reentrant features allow several tasks to execute identical sequences of code concurrently. A family of tasks was used to simulate the computers, where each computer asynchronously schedules and executes locally arriving tasks. If the new task cannot be guaranteed, synchronization/ communication with the other computers takes place through rendezvous. Using Ada, and its concurrent features, allows the simulation program of a multi-processor environment to be developed, debugged, and tested with minimal difficulty.

Introduction

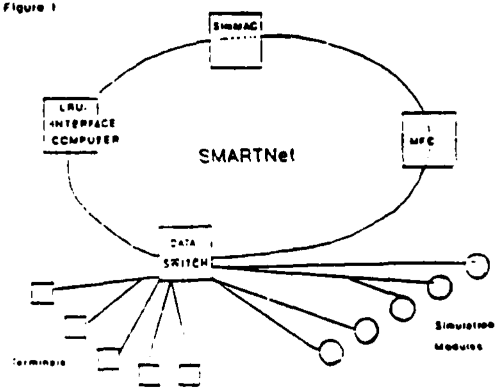
The Advanced Multi-Purpose Support Environment (AMPSE) is a generic, modular, and extendable system, designed to improve the overall supportability of weapon system software by reducing the costs and increasing the capability and flexibility of the real-time support environment. It provides a real-time simulation support environment for testing and evaluation of selected Embedded Computer System (ECS) Operational Flight Programs (OFPs). The support environment communicates with the OFP's via a dedicated ECS interface computer and a high speed real-time simulation network called SMARTNet (Shared Memory Architecture Real-Time Network). AMPSE's design is based on a modular building block approach to allow for system modifications and enhancements

The AMPSE simulation consists of a set of software models hosted on multiple distributed processors which provides the real-time simulation of the aircraft in an operational environment. The software models must be executed at specified rates in order to provide a realistic environment to the embedded computer software. Each model contains algorithms and/or data required to simulate a specific aircraft system or part of the external environment such as the aircraft's aerodynamics, avionics sensors, weapons systems and targets. These models are designed as separate software module that can be plugged into the simulation and easily moved among the simulation computers as requirements dictate.

The AMPSE provides the capability for multiple configurations ranging from a single LRU (Line Replaceable Units) with several models to a complete set of LRU's with only the aerodynamics and external environment models. Future enhancements would allow ECS models to be swapped with the aircraft's actual ECS or LRU executing their OFP's.

The AMPSE uses a RS-232 Data Switch, Ethernet, and SMARTNet for network communication within the system, see Figure 1.

Figure 1



The Data Switch provides manual operation and direct terminal to computer connection capabilities. Ethernet with DECNet protocol provides non-real time network for file transfers between the various simulation computers during the initialization and configuration modes.

The AMPSE operates in three modes:

1. Configuration: In this mode, the operator will define the hardware/software configuration of the simulation.

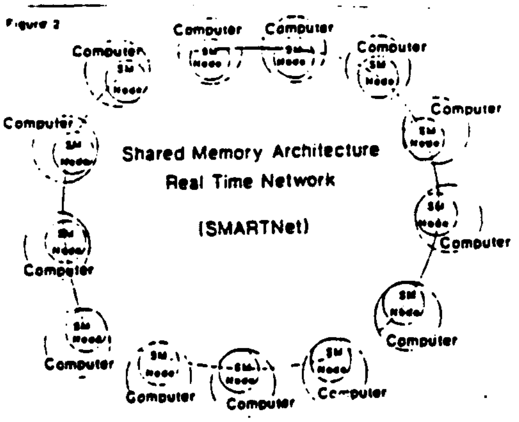
2. Initialization: In this mode, the operator will run the diagnostic routines, initialize the hardware, and load/initialize the proper software.

3. Execution: In this mode, the operator will perform real-time testing; interact directly with the real-time simulation, monitor simulation data and execution, and control the simulation testing.

During the execution mode, each computer on the network executes the software modules assigned at configuration/initialization. Each simulation computer has its own executive routines (local scheduler) to schedule and execute the modules. The local scheduler will execute the scheduled modules when an interrupt is received.

Communication with other computers on the network is performed via SMARTNet. The SMARTNet is a specially designed network, replicating shared memory, developed to support real-time multiprocessor simulation applications. With SMARTNet, each computer on the network has its own local copy of the shared memory. The shared memory is continuously updated over a high speed slot ring network, see figure 2.

Figure 2



Due to the fact that any computer on the network can have access to the data in shared memory computers can communicate by reading or writing to this memory.

Research Objectives

The current system executive (local scheduler) is working in Fortran on the MicroVaxes connected to Ethernet and SMARTNet. There is almost no task scheduling mechanism in its distributed executive. If there is one we can consider it only as static, at most. The models assigned to the simulation computers for execution were predetermined at system configuration, therefore the processing power of the computers are not fully utilized. In report [1], two dynamic task scheduling algorithms for the AMPSE were proposed.

These algorithms as attempt to fully utilize the available processing power by dynamically balancing the load among the computers. The first algorithm schedules the software modules by using a simple dynamic probabilistic algorithm. The second dynamic task scheduling algorithm is based on the availability of CPU resources.

The simulation of the second algorithm consists of 4 nodes connected in a slot-ring configuration with SMARTNet. There is a SMARTNet node and a local scheduler in each processor. In addition, there is a global scheduler for the whole system. There is also a SMARTNet node in the processor where the global scheduler resides.

The local scheduler is activated upon the arrival of a new task or in response to the bidding which is initiated by the global scheduler. The local scheduler determines if a new task can be inserted into the current System Task Table (each entry in the STT contains fields for Computer-Id, Task-Arrival-Time, Latest-Start-Time, Deadline, and Computation-Time) such that all previous tasks in the STT as well as the new task are guaranteed to execute. The latest start time is then computed and put into the corresponding entry in STT.

New tasks that cannot be guaranteed locally, or can only be accommodated at the expense of some previously guaranteed task are rejected by the local scheduler and are handed over to the global scheduler. The global scheduler then takes the necessary actions to transfer the rejected tasks to any alternate computer that may have the resources required to accept those tasks (using the bidding schemes as developed by Stankovic, J.A. Et al. [4,5,6,7]).

Results

The simulation of the algorithms presented in [1] was performed using the Ada programming language on a VAX 8530 under VMS [2].

Ada was chosen in the development of the simulation program primarily for its built-in facilities. Concurrent processes and their mechanism to communicate/ synchronize can be constructed in the Ada programming language by using its high-level language constructs. Also, Ada's run time system takes care of task scheduling and inter-task communication and synchronization. If the development language such as C, Pascal, or Fortran is used a multi-tasking executive must be written to handle the scheduling, communication and synchronization between tasks [3].

A family of tasks was used to represent parallel execution of the local executives, represented by:

```
local_executive : array(1..#nodes) of
                  local_executive_task;
```

This representation provides an easy way to tune the algorithm to the actual hardware environment.

In this simulation, it is shown that load balancing in a distributed system can be achieve by using a dynamic probabilistic algorithm, based on one proposed by Stankovic. This algorithm is easy to implement in Ada programming language, and is effective as the results of the simulation show [2]. Simulation results in an algorithm, showing that the average response times tend to decrease as the task interarrival time of the arriving nonperiodic tasks increases in proportion to their computational times. The simulation results of the algorithm also indicate that the CPU utilization is balanced among the four simulation computers, and that the system also guarantees the majority of these tasks with a shorter average response time tnan expected.

References

1. Liu, D.B., "Dynamic Task Scheduling for the Ada Distributed System Evaluation Testbed," Final Report USAF-UES Summer Faculty Research Program, August, 1989.

2. Liu, D.B., "Simulation of Dynamic Task Scheduling Algorithms for Ada Distributed System Evaluation Testbed," Final Report USAF-UES Summer Faculty Research Program, September, 1990.

3. Laird, J.D., "Real-Time Issues in Ada," Proceedings of the 1985 AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, CA, October 21-23, 1985.

4. Ramamritham, K., and Stankovic, J.A., "Dynamic Task Scheduling in Hard Real-Time Systems," Ieee Software, Vol. 1, July, 1984.

5. Sha, L., Lehoczy, J.P., and Rajkumar, R., "Task Scheduling in Distributed Real-Time Systems," IEEE, Proceedings, Industrial Electronics Conference, 1987.

6. Stankovic, J.A., and Ramamritham, K, "The Spring Kernal: A New Paradigm for Real-Time Operating Systems," Sixth IEEE Workshop on Real-Time Operating Systems and Software, SEI, Pittsburgh, PA, May, 1989.

7. Stankovic, J.A., Ramamritham, K., and Cheng, S., "Evaluation of a Bidding Algorithm for Hard Real-Time Distributed Systems," IEEE Transactions on Computers, Vol. C-34, No. 12, December, 1985.

Steven K. Iwohara is a Member of the Technical Staff at Rockwell International, Space Systems Division, Downey, California. He received his B.S. in Computer Science and Engineering from California State University, Long Beach and is currently pursuing a M.S. in Computer Science and Engineering at California State University, Long Beach.

Dar-Biau Liu is a Professor at California State University, Long Beach where he has been a faculty member since August 1986. He teaches graduate and undergraduate classes in Software Engineering, Distributed Computer Systems, Computing Theory and Programming Methods. His research interest include Software Reusability, Object-Oriented Design, and Dynamic Task Scheduling in Distributed Computer Systems. Before coming to California State University, Long Beach he was a faculty member at Old Dominion University, Norfolk, Va. Previously, he was a Staff Engineer at IBM Corp. and was a project manager at ITT Corp. He received a PH.D. in Applied Mathematics and Computer Science at the University of Wisconsin-Madison in 1972. Previously he had received a M.A. in Mathematics from Wayne State University, and a B.S. in Mathematics from National Taiwan Normal University. His current address is: Department of Computer Science and Engineering, California State University, Long Beach, Ca 90840.

FRODO

Far-Range Orbiting Defence Outpost

Charles F. Rose, III

ABSTRACT

This paper is concerned with the scheduling of resources in a simulated real-time system. It is demonstrated with a program application: a simulated orbital defence installation, FRODO. This simulated system is engaged in the rapid destruction of incoming enemy forces, and it needs to maximize its use of available resources to achieve its goal. Implemented using Ada's support of tasking and inter-process communications, FRODO demonstrates many of the issues which are associated with a multiprocessed application: the synchronization of events, the avoiding of resource contention or inefficient use of resources, as well as the problem of program deadlock.

WHAT IS FRODO?

FRODO (Far-Range Orbiting Defence Outpost) is a simulated defence installation. The primary goal of the simulated system is to defend the Earth against hypothetical incoming enemy forces. To accomplish this goal, the system must identify targets, decide which targets should be destroyed first, decide how many weapon resources should be allocated for a given target, and use the available weaponry to destroy these targets.

Programming applications such as FRODO, which can be broken down into discrete and mostly-independent units and which are real-time in their scope need to be developed in an environment which supports multitasking or parallel computing. Ada's comprehensive support of multiprocessing and inter-process communication make it a logical choice as the language of development for this type of application.

The simulation involves the reaction by the FRODO system to the presence of enemy forces which it must destroy. The enemy forces advance to attack the Earth and the installation and FRODO must react to affect their destruction. As a simulation, certain rules govern the actions of the objects being simulated and these rules will be detailed for completeness.

FRODO itself is conceived of as an orbiting defence installation with a given number of weapons of two different types. This was implemented so as to make more complicated (and more realistic) the multi-process communication found in the system. One type of weapon inflicts less damage but requires less down-time before it can be fired again and the other type inflicts a greater amount of damage but takes longer to re-attain readiness after firing. The amount of these weapons available to the installation is arbitrary and is defined by the program. Also, the logic of the system will handle the different numbers of weapons and allocate them accordingly. While FRODO is most logically an orbiting station, for purposes of this simulation, it can be thought of as having a stationary position some distance from the Earth.

The enemy's forces travel towards the Earth from some distant point and must bypass the protective shield offered by the FRODO installation. Whenever possible, they use their resources to attack FRODO and the Earth. These targets will become visible when they are within some given range dependant upon what type of enemy ship they are. Like all other aspects of the simulation, these ranges can be changed in order to alter the logic of the program.

Enemy forces fall into a few categories and their particular missions are defined by these categories. These types include cruisers, bombers, fighters, and missiles. Cruisers are the strongest of these enemies, being the least susceptible to damage from attack, having the ability to withstand more damage, and having the largest arsenals at their disposal. Bombers are weaker than cruisers and fighters are weaker still. The ships follow different patterns of attack which are defined by the category into which a given target falls. Fighters always attack FRODO, bombers always attack the Earth, and cruisers attack either the Earth or FRODO. Two other types of targets eventuate during the course of the running of the simulation. These are Earth- and FRODO-bound missiles and they are launched by the different enemy targets. Missile targets are never present at the start of a given simulation.

GAME LOGIC

Since this program is essentially a simulator, there exists some logic dedicated to the resolution of things outside the scope of the actual application. This includes the logic to resolve attacks made by either side, as well as the logic which decides if a win or loss has been achieved for a given scenario. This logic is defined as the game logic for the simulation.

This game logic was designed to take a minimal amount of processing time and program space while maintaining as much realism as possible in the simulation so as to accurately stimulate yet not interfere with the FRODO system. Wherever possible, the game logic was placed in the parts of the program as removed as possible from the main tasks of the FRODO system, i.e. those processes dedicated to deciding which targets to attack, and those which control the weapon systems of the system.

Before a scenario is run using the FRODO simulator, its data is generated and its parameters are set. The strengths of the enemy forces and the other parameters regarding their initial positions, velocity, and susceptibility to damage and detection are generated by the scenario generating routine or are defined in the game logic of the program and can be easily modified to affect the actual actions taken by the system at run-time.

There are definite criteria detailing whether the FRODO simulation has been effective in completing its task. It has been successful (resulting in a win) if all enemy forces have been destroyed and no enemy attacks affected the Earth. If at least one enemy attack hit the Earth, i.e. an Earth-bound missile was not destroyed before reaching the planet, then the simulation has been unsuccessful and results in a loss.

THE FRODO SYSTEM

FRODO is divided into a number of discrete and semi-independent concurrent tasks. Each of these processes performs a portion of the actions required by the system as a whole and a process will occasionally rendezvous with another to coordinate some event, i.e. firing a weapon. These processes were implemented using Ada tasking and the rendezvous between the processes were implemented using Ada's support for inter-process communications under tasking.

FRODO has three unique and two non-unique types of tasks which compromise its system. The unique tasks are DECIDE, CONTROL, and SCAN. In addition there are a number of non-unique tasks in the system, one for each of the weapons. The number of weapon tasks corresponds to the number of weapons defined for the system. An overview of these tasks is provided here and a more detailed study of their functionality follows.

DECIDE is the main task governing the actions of the other tasks. It is the one which decides which targets to attack and what types of weapons to dedicate to any given attack. SCAN's functions include the tracking of targets and updating of target information. CONTROL keeps track of which weapons are currently available and handles requests for the use of those weapons. The weapon tasks control the firing of a weapon as well as the signalling of whether that weapon is ready or not.

These tasks have different levels of autonomy and are governed accordingly. Certain actions, i.e. firing a weapon, require the synchronization of many tasks so as not to compromise the accuracy of system-wide information and so require a great deal of inter-process communications to achieve this synchronization, while others, such as tracking the targets, require little governance or inter-process communication.

SYSTEM-WIDE INFORMATION: TARGET DATA AND STATUS FLAGS

In order for FRODO to complete its tasks, it must maintain some information which is accessible to many parts of the system. This data includes the information on the targets currently being tracked and the various status flags which are maintained by some of the tasks.

For every target being tracked, certain information must be kept. This includes its location, speed, and what type of target it is (cruiser, bomber, etc.). In addition to the scanned information stored with the target, there is some derived information stored, such as ETA and the priority of a target. This information is periodically updated and accessed in order to make decisions concerning which targets to are to be attacked.

The status flags kept include which and how many weapons are available for any given weapon type. This information is updated by the weapon and weapon control processes and is used by the decision process. It is used to decide how many weapon resources to devote to any given attack.

THE DECIDE PROCESS

The DECIDE task is the coordinating task for the whole system. It must initialize and terminate all the other tasks, in addition to allocating the weaponry for the speedy completion of the mission. DECIDE determines which targets are to be attacked first based on the current bias of the system (Protect Earth, Protect Self, or Destroy Enemies) and also decides if the bias of the system is to be changed.

```
task type DECIDE is — define the entry points into the task
  entry INITIALIZE;
  entry FIRST_SCAN;
  entry CONTROL_READY;
  entry WEAPONS_LAUNCHED(weapon: in weapon_type);
  entry STOP_FUNCTION;
  entry SCAN_DOWN;
  entry CONTROL_DOWN;
end DECIDE;

task body DECIDE is
  accept INITIALIZE; --- Don't do anything until initialized
  initialize the CONTROL and SCAN sub processes
  accept FIRST_SCAN; --SCAN has confirmed its initialization
  accept CONTROL_READY; -- CONTROL has initialized
                                    -- itself and its subprocesses

  while not instructed to stop functioning loop
    select
      accept STOP_FUNCTION; -- stop looping
     or
       calculate which of the available targets to attack
         and allocate the necessary resources to do so with
         a call to the CONTROL process. Make sure the proper
         synchronization protocols have been followed to avoid
         program deadlock or the compromise of the integrity of
         system-wide status information
    end select;
  end loop;

  terminate the CONTROL process
  terminate the SCAN process
  accept CONTROL_DOWN; — CONTROL has confirmed
    its termination
  accept SCAN_DOWN; — SCAN has confirmed its termination
end DECIDE
```

Once the process has been initialized (discussed in bringing up the system) and it has initialized and received confirmation of initialization from its dependant subprocesses (CONTROL, SCAN, and the weapons), it will begin to repeatedly decide which enemies are to be attacked and what resources will be allocated for these attacks until it is instructed to stop. This signal must come from an outside source (when the game logic determines that the conditions for a win or loss have been met).

The firing of a weapon by FRODO involves the rendezvous of the DECIDE and CONTROL processes and the weapon processes in question through the CONTROL process. This is done in order to prevent the contentious use of resources and to maintain the integrity of the system-wide information, i.e. how many weapons are available and which individual weapon systems are ready to fire. The DECIDE process, upon requesting a weapon to be fired signals CONTROL of this request and waits for confirmation from CONTROL that the action has been accomplished. CONTROL allocates its resources and signals a request to those weapon processes and waits for confirmation of the processing of that signal by those weapon processes. CONTROL then signals DECIDE of its completion, thus preventing unwanted concurrency during this critical section of the logic. If this is not done, program deadlock may occur when DECIDE signals CONTROL to use resources that it does not currently possess but thinks it does.

When DECIDE has received the instruction to cease functionality, it will first bring down the SCAN and CONTROL processes.

THE CONTROL PROCESS

Control is the process which handles requests for the allocation of weaponry for the attack of a target. In addition, this process must initialize the weapon systems, keep track of which and how many weapons of a given type are currently available, and must gracefully terminate all the weapon tasks which are controlled by the CONTROL process.

```
task type CONTROL is -- define the entry points into the process
  entry INITIALIZE;
  entry FIRE_WEAPON(weapon: in weapon_type;
                    target_ID: target_ID_type;
                    attack_with: in weapon_amount_type);
  entry WEAPON_READY(weapon: in weapon_type;
                                weapon_ID: in weapon_ID_type);
   entry WEAPON_DOWN(weapon: in weapon_type;
                                weapon_ID: in weapon_ID_type);
   entry STOP_FIRING;
end CONTROL;

task body CONTROL is
   accept INITIALIZE; -- Don't do anything until initialized
   initialize all the weapons which CONTROL coordinates;
   accept confirmation of readiness from each of the   initialized
      weapons;
   signal DECIDE of readiness;

   while not instructed to stop firing loop
      select
         accept WEAPON_READY(weapon: in weapon_type;
                                    weapon_ID: in weapon_ID_type);  -
                                   --a weapon has come on line
         update appropriate status information
      or
         accept WEAPON_DOWN(weapon: in weapon_type;
                                   weapon_ID: in weapon_ID_type);
                                   -- a weapon has gone off line
         update appropriate status information
      or
         accept FIRE_WEAPON(weapon: in weapon_type;
                               target_ID: target_ID_type;
                               attack_with: in weapon_amount_type);
         allocate the appropriate weapons, instruct them to fire,
            and receive confirmation from those weapons that the
            requested action has been performed and then inform
            DECIDE the action has been carried out;
      or
         accept STOP_FIRING; --cease looping at this point
      end select;
   end loop;

   terminate the weapon processes;
   wait for confirmation of that termination;
   signal DECIDE that termination is complete
end CONTROL;
```

Once control has been initialized, it initializes all the weapon tasks dependant upon it and waits to receive confirmation of their initialization. It then sets the system-wide information to reflect the current state of readiness in the system. Then it signals DECIDE of its readiness, after which CONTROL begins to perform its main task, the coordination of resource (weapon) allocation.

Upon receiving a call to fire a weapon of a given type, CONTROL has the task of deciding which weapon is to be fired. It signals the weapon system to fire, receives confirmation of this action, and then signals DECIDE of the completion of a fire procedure, after which CONTROL continues to process.

In addition to firing a weapon, CONTROL must keep track of which weapons are currently available and quickly process readiness signals from the weapons so as to maximize the amount of up-time for a given weapon.

Once receiving the signal to stop firing, the CONTROL process must bring the weapon processes to a graceful halt.

THE SCAN PROCESS

SCAN is the task which periodically updates the target information of the system. It is the most independent of the tasks; only interacting with DECIDE to start and stop its processing of target data. It works with the target data set which is in then accessed by the DECIDE process (and some game-logic routines).

```
task type SCAN is  -- define the entry points into the process
   entry INITIALIZE;
   entry STOP_SCAN;
end SCAN;

task body SCAN is
   accept INITIALIZATION; -- Don't do anything until initialized
   complete an initial scan;
   signal readiness to DECIDE;
   while not instructed to stop scanning loop
      select
         accept STOP_SCAN;  -- stop looping at this point
      or
         delay for a specified amount of time and then update the
            target information.
      end select;
   end loop;
   signal DECIDE that SCAN is down;
end SCAN;
```

Once initialized, SCAN delays for a specified amount of time and then updates the list of targets. It continues to do so until instructed to stop scanning.

Because of its location and independence in the system, SCAN proved to be one of those processing areas in which much of the game logic could be hidden. Along with updating the target data, the update target-list routine executes much of the game logic.

THE WEAPON TASKS

The weapon tasks control the actual firing of the weaponry in FRODO. They are the only tasks which are non-unique in the system and the weapon banks are implemented as arrays of these tasks.

```
task type WEAPON is
   entry INITIALIZATION(weapon_ID: weapon_ID_type);
   entry FIRE(target_ID: target_ID_type);
   entry CEASE_READINESS;
end WEAPON;
```

```
task body WEAPON is
  accept INITIALIZATION(Weapon_ID; weapon_ID_type);
                                        -- Don't do anything until initialized
  signal readiness to CONTROL;
  while not instructed to cease readiness loop
    select
      accept FIRE(target_ID:target_ID_type);
      signal down-ness to CONTROL;
      resolve the attack;
      delay while the weapon has to wait;
      signal readiness to CONTROL;
    or
      accept CEASE_READINESS; -- stop looping at this point
    end select;
   end loop;

  signal CONTROL that the weapon is down;
end WEAPON;
```

After initialization of the individual weapon task, the process accepts either a message to fire or cease readiness. It processes requests to fire until told to stop firing. During initialization, each weapon process is given a weapon ID which it uses when it signals CONTROL so that CONTROL will have the knowledge of which particular weapon is making an entry.

Once a signal to fire is received, the weapon first signals the fact that it is going off-line to CONTROL. It then fires and waits for the appropriate period of down-time before signaling its readiness to CONTROL. The length of the down-time is defined by the program for a given weapon type.

STARTING AND STOPPING THE SYSTEM

Since FRODO consists of many independently running tasks, they must be brought to full functionality and terminated in a specific sequence to insure that no unexpected results occur during processing.

As DECIDE is the coordinating process for the whole system, it is DECIDE which is instructed to initialize the system as a whole. Once it has been initialized, it brings up its dependant sub-processes, CONTROL and SCAN, and waits for confirmation of their initializations.

The dependant subprocesses CONTROL, SCAN, and the weapon processes (dependant to CONTROL) are all brought up before DECIDE is permitted to enter its main function loop, the continuous allocation of weapons for the attack of enemy targets.

Terminating the system follows a similar pattern. Processes accept signals instructing them to terminate and they signal any dependant processes to terminate first, then accept confirmation of those terminations, then signal confirmation of their own terminations. As before, termination of FRODO is done by requesting termination of the controlling task, DECIDE.

RESOURCE ALLOCATION, PRIORITY OF TARGETS, AND BIAS OF THE SYSTEM

Whenever the overhead of the system has been processed (flag updates), FRODO attempts to attack enemy targets. Depending on the relative priority of the targets and the current bias of the system, as well as the type or target being slated for attack, a different amount of resources will be allocated for that target.

The rules for determining which targets are to be attacked and which resources are to be allocated are determined on type, bias, and closeness of a target. Cruisers, bombers, fighters, Earth- and FRODO-bound missiles are all given a relative weighting which is modified by the bias of the system. These weightings can be changed to adjust the actions that FRODO will take in a given situation.

The closeness of a target is equal to its ETA and it factors into the priority of a target each time it crosses from one band of priority to another based on its ETA. These bands can be defined at different locations to increase or decrease reliance on closeness of a target in determining its priority in the attack queue. If a situation arises in which two targets hold equal priority, the closest one will be attacked first.

The system can be biased in its attack patterns based on the current mission of the system. The missions that the system can be performing include: protecting the Earth, protecting the defence installation, and destroying the enemy forces. Various situations can cause the mission to change. For example, if the Earth is currently under attack from Earth-bound missiles, the mission will change from whatever value it currently holds to be that of protecting the Earth. While the bias of the system can greatly influence the order in which targets are slated for attack, it can at no time exclude a target from being considered.

RESOURCE CONTENTION AND DEADLOCK

In a system of this type many situations can potentially arise in which there will be a contentious use of resources or in which the processes will be in a state of deadlock.

One instance of deadlock can occur in a situation in which the processing of any readiness signals is relegated to a subordinate position in the CONTROL process. If one type of weapon readiness entry is given a lower priority than another readiness entry for a weapon of another type, then a period of process lockout will occur during times of frequent weapon firing. This causes weapon processes of one type to have to wait to signal their readiness until all requests to fire are completed, which makes these weapons virtually useless:

Weapon A of type α is fired
Weapon A of type α goes off-line
Weapon B of type α is fired
Weapon B of type α goes off-line
Weapon A of type β is fired
Weapon A of type α issues a request to be brought back on-line
Weapon A of type α is brought back on line
Weapon B of type α issues a request to be brought back on-line
Weapon A of type α is fired
Weapon A of type α goes off-line
Weapon A of type β issues a request to be brought back on-line
Weapon B of type α is brought back on-line
Weapon A of type α issues a request to be brought back on-line
Weapon B of type α is fired
Weapon B of type α goes off-line
Weapon A of type α is brought back on line

Even though two weapon systems were ready to be brought back on-line, only one was permitted to do so and the weapon of type α was favored over the weapon of type β even though the weapon of type β had issued its request prior to that of the weapon of type α. So a system in which a weapon of any type could be brought back on-line with a single type of entry is used. This insures that no given class of weaponry is locked-out in favor of another class.

Another instance of deadlock can occur if the timing of critical events is not handled with great care. This can occur because of the contentious use of resources arising if the current state of the system as reported by the status-flags and the actual state determined by the tasks of the system no longer coincide. This can happen because of improper synchronization of code during a critical section. The primary example of this in the FRODO system is found in the procedure of firing a weapon. If DECIDE makes a request to fire a weapon without waiting for confirmation, there is no assurance that the appropriate status-flags will be set before DECIDE targets another enemy. At this point DECIDE will have fewer weapons than it thinks it possesses. This situation is avoided by the synchronization of the DECIDE and CONTROL tasks (and the weapon process through CONTROL) during a weapon firing process.

CONCLUSION

FRODO (Far-Range Orbiting Defence Outpost) is a simulated defence installation which is charged with the defence of the Earth against the forces of a hypothetical enemy. It was developed using Ada's support of multiprocessing and inter-process communications with Ada tasking. The system is divided into a number of separate processes working together to complete the mission, each of which is represented by an Ada task. Coordination of these processes is handled with great care in order to maintain the integrity of system-wide information, as well as to avoid program deadlock or inefficient use of resources. This is accomplished using Ada's support of inter-process communications.

BIOGRAPHICAL

Charles F. Rose, III is an honors student attending Trenton State College in New Jersey as a computer science major. He is currently a junior and is planning to pursue graduate study in the computing sciences. He is a member of the programming team at the college in addition to being a member of Upsilon Pi Epsilon, the computer science honor fraternity. In addition, he is involved in the design and development of technical support programs for the Integrated Systems Division of Computer Sciences Corporation in Moorestown, New Jersey.

Charles F. Rose, III
809 Halliard Avenue
Beachwood NJ 08722

ADA, MODULARITY AND INTEGRATION

Kenneth L. Ivey

The University of Mississippi

ABSTRACT

This paper examines the features of Ada which provide for modularity, separate compilation, and the separation of specifications and bodies of programs. Then, in light of the recent experiences in writing a large statistical math package, it discusses the value of the various features of the language in enhancing the correctness and consistency of the final project. Suggestions are made for ways of using the language features at each of the software lifecycle stages to enhance programmer productivity and insure program correctness.

Section 1: INTRODUCTION

The project which this paper describes involved an agreement between a software engineering class at the University of Mississippi and IBM Federal Sector Division. The agreement required us to write a Statistical Plot Package in Ada on an IBM 3084-QXC running VM/XA using CMS version 5.12. The specifications were provided to the class by IBM FSD. They also provided and required their in-house programming standards and DOD-STD-2167 and DOD-STD-2167A. The package was to include the following:

1. A Correlation Function
2. Three Curve Fit Functions (Curve Fit, Time, Step)
3. A Histogram Function
4. A Plot Function
5. A Statistical Sampling Function
6. A Set of functions to provide statistical summaries
7. A Sort Function
8. A Tabular Function.

The class was divided into five groups, four of which would write code and one group which would be in charge of the integration of the separate pieces written by each of the different groups. Each group was given an account on the IBM 3084 where all development was to be done. Class meetings were held twice a week for one hour and fifteen minutes. The class meetings consisted of lectures on the Ada language itself in the beginning. Later lectures were on the Software Engineering Lifecycle. Finally discussions shifted to reviews of the requirement documents provided to the class by IBM and then to the design issues themselves.

IBM provided the class with a wide range of documents. Included in these documents were the "Ada-Based Process Design Language (PDL/Ada-2)", DOD-STD-2167 ,DOD-STD-2167A and complete specifications about each of the statistical functions our package was to support. The actual design documents were to be created using IBM's PDL/Ada-2. The package that we were to provide was part of a much larger system which we knew nothing about. IBM provided us with a chart showing us just how our part fit into the total project but at no time were we aware of just how our package would be used.

Section 2: LIFECYCLE PHASES

Section 2.1: REQUIREMENTS REVIEW

The first official task of the class was a requirements review held during one entire class period and continuing on for most of the afternoon. During this meeting we were allowed to quiz two representatives from IBM FSD about our understanding of the requirements and to clear up any ambiguities which we perceived at that point. We had no formal document to present to IBM at this stage. We only had to be sure that we knew just what they wanted. After the Requirements Review we began in earnest to develop our preliminary design.

Section 2.2: PRELIMINARY DESIGN PHASE

At this stage of the Software Engineering Lifecycle, our design document was completely written in IBM's PDL/Ada-2 which resulted in fully compilable and commented code.

```
-<
- provide a statistical summary using the data supplied by
- the operator
->

procedure STATISTICAL_SUMMARY
        (FILENAME      : in  FILE_NAME_TYPE;
         FIELD         : in  FIELD_NAME_TYPE;
         FIELD_NAME    : in  FIELD_ID_TYPE;
         TITLE         : in  GRAPH_LABEL_TYPE;
         OUTPUT_DEVICE : in  OUTPUT_DEVICE_TYPE);

        -<
        - using the data supplied in FILENAME, create a
        - statistical summary of the data in FIELD and send to
        - OUTPUT_DEVICE.
        ->
```

(It should be mentioned here that there were supporting documents which were not written in Ada but they served as documentation only) The design document primarily consisted of heavily commented Ada specifications. The specification is the part of the program unit that stipulates the interfaces with other program units. It defines the external characteristics of that particular unit. Only calls to procedures listed in the Specifications are allowed. No access is allowed to anything not specifically mentioned in the formal program unit Specification. The body is the second part of a program unit. The body provides for the implementation of the series of operations that the unit performs. The separation of the Specification from the Body in a programming unit enforces programmer abstractions and ensures information hiding.

Section 2.3: DETAILED DESIGN PHASE

Let us return to the project. After a formal review by IBM and their acceptance of the Preliminary Design Document, we then progressed into the Detailed Design phase. This involved adding Ada specifications of the bodies for each specification declared in the Preliminary Design Document and adding, in place of sequence_of_statements, actual stubbed out loops and logic structures in the form of comments in each body. This process followed the standards dictated in DOD-STD-2167A.

```
-<
- using the data supplied in FILENAME, create a
- statistical summary of the data in FIELD and send to
- OUTPUT_DEVICE.
->

        procedure STATISTICAL_SUMMARY
                (FILENAME      : in  FILE_NAME_TYPE;
                 FIELD         : in  FIELD_NAME_TYPE;
                 FIELD_NAME    : in  FIELD_ID_TYPE;
                 TITLE         : in  GRAPH_LABEL_TYPE;
                 OUTPUT_DEVICE : in  OUTPUT_DEVICE_TYPE)

-<
- ( Errors in peramemters or data   -> RAISE
                                  Exception TBD),
- ( NOT END_OF_FILE ( FILENAME )  ->
-   Exist TEMPORARY_RECORD : record in ( FILENAME : TBD )
-   Such that TEMPORARY_RECORD := READ( FILENAME ),
-   CALCULATIONS (TEMPORARY_RECORD.FIELD )
-
- | TRUE                        ->
-   CLOSE ( FILENAME ),
-   OUTPUT ( TITLE, FIELD_NAME, RESULTS (CALCULATIONS) )
-     to OUTPUT_DEVICE
- )
;
```

Although the DOD-STD-2167A format could be used with most any methodology, it directly maps onto Ada. But even at this level we used our Preliminary design document to create this new (required) Detailed Design Document. This building process would continue throughout the entire project. This is an important issue because each phase is just a continuation of the previous one. At no point did we have to take some chart (i.e. HIPO, Data-Flow Diagrams) and convert it into Ada code. The framework of the project became easier to see when you always built on previous design documents when creating the next document in the lifecycle process.

Section 2.4: CODING PHASE

Next came the coding phase. This was one of the simplest stages, because of the development lifecycle that we had been using. All of the control statements were already in place at this point in the form of stubs and had been successfully compiled at each of the previous stages. All we had to do was take this Detailed Design Document, which now consisted of Ada Specifications and Ada Bodies, where the Body was as described earlier, and convert all the comments into actual Ada code.

I remind you that the actual coding at each stage was done on four different accounts. The individual groups were responsible, at each stage, for providing the integrating group with thւ appropriate design document pieces for the area of the project for which they were responsible, e.g. one group was responsible for all of the plot, histogram, and sampling procedures. At no time during the coding process was the entire code for the project located on a single user's account. It became the task of the integrating group to develop tools and procedures, on the project's primary account, to allow for a rapid and smooth integration of the complete Statistical Plot Package.

Section 3: INTEGRATION

Section 3.1: PRELIMINARY DESIGN PHASE

This first attempt at integrating all the pieces of this project came at the end of the Preliminary Design Phase. Each of the four different groups had prepared Preliminary Design Documents for each of the functions which had been assigned to them. Because of time constraints, the integration team only had enough time to run the code through the parser. This eliminated all the syntax errors, but several other errors still existed. The design was sent on schedule to IBM where they compiled it on their Rational System, but to our horror, when the report of the errors returned from IBM, there were a total of twenty-nine. Most of the errors had to do with our not adhering to their design specification package called "SYSTEM" which contained all the type definitions required for the project e.g. instead of INTEGER we were to use INTEGER_TYPE which was a bounded range of integers dictated by IBM FSD.

Section 3.2: DETAILED DESIGN PHASE

When the time came for the integration of Phase two, the Detailed Design Document, a more solid, stepwise plan had been developed. The integration team first put all the specifications together and successfully compiled them. This integration step had to be repeated at this phase because changes had been made to the interfaces after the review of the Preliminary Design. This was not a problem this time, because each group had taken more time to make sure their specifications were correct. Next the Bodies were added. This was a fairly straightforward process. Only one body would be added and then immediately compiled and tested. Once it was working, another body would be added and so on. One problem we ran into at this stage was making sure the types between the interfaces were consistent, i.e. INTEGER instead of INTEGER_TYPE. The other problem at this point was making sure the specification of the body conformed to the specification of the declaration. It should be pointed out here again, that as we were progressing through this lifecycle, we were getting closer and closer to the final code. Using PDL/ADA-2 did not involve any charts or diagrams that had to be converted. Both DOD-STD-2167 and DOD-STD-2167A did require matrices and charts, but these were done as supporting documents to the actual design and not as actual design documents. All of the actual design documents were compiled and commented Ada code.

Section 3.3: CODING PHASE

The final phase of integration came when all the coding had been completed. This was the most difficult and time-consuming phase so far. Again here a plan had been devised to be used during this integration period. As in phase two, we first integrated all of the specifications into one physical file and continued to work with them until they successfully compiled. Then the bodies of the procedures were stubbed out one at a time using Ada's "IS SEPARATE" form.
This would be in the PACKAGE BODY:

```
procedure STATISTICAL_SUMMARY
                    (FILENAME      : in  FILE_NAME_TYPE;
                     FIELD         : in  FIELD_NAME_TYPE;
                     FIELD_NAME    : in  FIELD_ID_TYPE;
                     TITLE         : in  GRAPH_LABEL_TYPE;
                     OUTPUT_DEVICE : in  OUTPUT_DEVICE_TYPE)
        is SEPARATE;
```

This would be in a PHYSICALLY SEPARATE file:

```
separate (PACKAGE_NAME)
procedure STATISTICAL_SUMMARY
                    (FILENAME      : in  FILE_NAME_TYPE;
                     FIELD         : in  FIELD_NAME_TYPE;
                     FIELD_NAME    : in  FIELD_ID_TYPE;
                     TITLE         : in  GRAPH_LABEL_TYPE;
                     OUTPUT_DEVICE : in  OUTPUT_DEVICE_TYPE) is
        begin
 – actual code
 – on these lines
end  STATISTICAL_SUMMARY;
```

This allowed the executable code of the procedures to be in a physically separate file. The referencing environment for that code remained in the primary body as though the actual code was inline in that body. One of the biggest advantages in doing this was, that though the size of the Body was substantial it never, during integration, got excessively large, this made edits during the integration phase much easier. Also the actual code for the procedures were in small files. This localized any errors encountered. The size of the files made a big difference since the final product was about seven thousand lines of actual code, not including comments and headers. Compiling or searching a few hundred lines of code is much easier and much quicker than seven thousand. Only when a body had been successfully compiled was another body stubbed out. Finally after all the procedure bodies had been successfully compiled the "IS SEPARATE" 's were removed and the actual code put inline. This final step caused no problems because when the "IS SEPARATE" form is used, it is as though the actual code is physically in line.

Section 4: PROBLEMS IN FINAL INTEGRATION

One of the first problems we encountered integrating this final phase was in interface consistency. This mainly involved calls to procedures in our I/O package that were changed at the last minute and the communication of these changes was sometimes not received by all groups.

Another set of problems developed involving the propagation of user-defined errors. Any user-defined error defined in the I/O package that needed to be propagated up so it could be trapped by any program "WITHing" our package would first have to be renamed in our specification using Ada's "RENAMES" form.

```
NAME_ERROR : exception renames
STATPLOT_IO.NAME_ERROR;
```

Section 5: TESTING

The final step for the integration team was to test each function. Each group was responsible for testing its procedures for correctness and accuracy. The testing by the integration team was just a check to make sure each procedure still worked in its new environment. A test program was written by the integration team to serve as a demonstration to IBM representatives at the delivery meeting. The test program simply made calls to all of our functions in the Statistical Plot Package to demonstrate the usage and to show sample output, i.e. screen, files, and printouts.

Section 6: CONCLUSIONS

There are some features of Ada that help to insure modularity and enhance the correctness and consistency of the final project. Because Ada provides, in the use of a specification and Body in the structure of each of its programming units, a means of separating the user interface from the implementation details, it was a totally straightforward procedure to use only Ada specifications to describe the overall project at the Preliminary Design stage. We knew at this time what the interfaces to our functions would be so it would be a simple process to create the specifications. We knew at Preliminary Design that each of the functions would need to call some procedures to deal with I/O. We could implement this as calls to "Mystical I/O" and leave it at that, compiling the specification of "Mystical I/O" without any information about how the I/O would finally be implemented. The entire project could then be checked for consistency at each level using only Ada Tools.

A Package in Ada serves as an abstraction mechanism and allows the programmer to fully capture all of the program units and type specifications required to implement a particular programming abstraction. For instance, one might have a package called "STACK" which contained all of the functions and procedures required to implement a stack. This Ada feature allowed us to use a package called "SYSTEM DESIGN" provided by IBM to do all data definitions. That way all subcontractors on the project, not just us, could use the same data descriptors and avoid any type conflicts. We simply imported the package "SYSTEM DESIGN" at every stage of the design using an Ada "WITH" statement.

Mr. Kenneth L. Ivey is currently an undergraduate student at the University of Mississippi. He will graduate in May with a Bachelor of Arts in Computer Science. A senior from Jackson, Miss., he is president of the Student Chapter of the ACM. For the project reported on in the paper he served as the director of the integrating group. He plans on beginning his graduate work in Computer Science in May. He may be reached at: The Department of Computer and Information Science, 302 Weir Hall, University, MS 38677. ken@cs.OLEMISS.edu

Paper No. ____________

RELEASE STATEMENT

PAPER TITLE On the Use of Ada in Mathematical Applications

AUTHOR(S) Seth M. Lowrey

COMPANY The University ofMississippi

I have agreed to appear at the Ninth Annual National Conference on Ada Technology (ANCOAT) which is sponsored by the Annual National Conference on Software Technology, Inc. (ANCOST). I recognize that any paper or written document that I may submit in conjunction with my appearance at the Conference may be bound into the written Proceedings for the Conference. Therefore, the paper(s) is approved for public release and sale. Its distribution is unlimited. I agree that ANCOAT has no responsibility for the content of any paper I may submit, or for the content of my oral statement made at the Conference. I agree to release ANCOAT from any responsibility arising from the content of any such paper or speech. I further recognize that ANCOAT has no responsibility for the content of any other paper or speech submitted or made by any third party, unless ANCOAT specifically endorses such content, and I accordingly release ANCOAT from any responsibility arising from such papers or speeches.

SIGNATURE: Seth M.

Seth M. Lowrey

On the Use of Ada in Mathematical Applications

Seth M. Lowrey

The University of Mississippi

Abstract

In many applications the programming language Ada is used to perform mathematical and statistical calculations. There is some concern that programmers of these applications are not aware of the irregular problems which arise from transporting between unlike machines those programs that use floating point numbers and variables. Due to the rigorous overall specifications of validated Ada implementations, this problem appears to be a minor one since corrections can be made that are portable among all implementations; yet the chief danger lies in the unawareness of the programmer that there are limits to precision floating point manipulation in Ada implementations on all machines. These limits will be addressed as the method of encoding floating point numbers by validated Ada implementations and type conversions to and from floating point types are observed. Some general functions and their handling of floating point types will be examined through the use of a contrived example, demonstrating the pitfalls of precision programming in Ada using floating point types and the dangers of ignorant reliance on Ada's predeclared floating point types. It will be shown that, with proper precautions and specific portable error testing methods, floating point arithmetic may be effectively employed with greater regularity.

It is common knowledge among most programmers that floating point programming in most languages is unreliable and inaccurate to a certain degree. The fact that some numbers representable in some numeric systems are impossible to represent in others has haunted mathematicians as well as high-level language programmers. As a simple example, the value of one-third is still impossible to represent accurately in all but fractional number systems. A decimal representation can only show the repeated fractional part, "0.3333333 . . . ." Of late there has been progress in the standardization of floating point representations and a strong effort to create a sense of awareness about the traps of using language-specific facilities for floating point arithmetic.

There has been much recent progress in the development of floating point accuracy in the language Ada. Its emphasis on portability provides a solid platform on which to build accurate arithmetical functions which may be reused to maximum potential. Unfortunately many programmers depend blindly on the built-in floating point operations in Ada without recognizing its hidden inaccuracies because of Ada's specifications of "safe numbers," "model numbers," and "model intervals." The definition of the language states plainly that, "in the general case, division does not yield model numbers and in consequence one cannot assume that (1.0/X)*X = 1.0." (Ada L.R.M., 4-21) In fact, Ada provides nothing that causes its mathematical functions to be anymore accurate than those of other programming languages. Ada only promises (fairly) strict portability of its inaccuracies. This is odd, since Ada was designed as "a language with considerable expressive power covering a wide application domain." It appears that mathematical application were left out of this domain, and because of this now some programs for sensitive mathematical applications may be programmed without concern for errors propagated through use of Ada's basic arithmetical functions.

There is a definite problem here that we can hope Ada programmers are conscious of, but efforts must be increased to make sure of this. First we must have some knowledge of what makes Ada's floating point types portable. In Ada precision may be declared by the programmer to a certain number of decimal places if desired, or standard floating point types may be used for easier expression of operations. Numbers represented by either type method are internally represented according to a few basic axioms dealing with model intervals. For any real type definition a set of model numbers is defined which contains numbers which are completely representable on the machine for the current Ada implementation. The model numbers do not necessarily include all numbers representable on the machine, but they guarantee an accuracy on the specific type definition within bounds called model intervals. A model interval is any interval bounded by two model numbers. In an operation which results in a real subtype, the following statement insures accuracy to the specified degree no matter what platform is used:

> **The resulting model interval is the smallest model interval (of the result subtype) that includes the minimum and the maximum of all the values obtained by applying the (exact) mathematical operation, when each operand is given any value of the model interval (of the operand subtype) defined for the operand. (Ada L.R.M., 4-20)**

Thus the model interval of the final result will be related to the exact mathematical result in that the bounds of the model interval will also be bounds of the result. This is a strong statement for portability since specified precision in one Ada implementation will correspond in general to the same specified precision in other Ada implementations. This also demonstrates that arithmetical error for floating point operations will not exceed certain bounds, but it does not give the programmer a facility by which he may determine a more precise final result when intermediate results have digressed due to errors such as machine rounding errors.

The programmer of Ada programs obviously must be made aware that all floating point precision problems have not been rectified because of the portability of the Ada language. Other errors which arise in simple use of floating point numbers are comparison errors and type conversion errors which may have drastic effects, especially in statistical applications. For example, in a recently completed statistical software project, the Median and Upper and Lower Quartiles of a list of numeric values were required. In determining the Lower Quartile for N numericvalues, if N/4 has no fractional part, i.e., it is an integer, then the lower quartile is equal to (VALUE[N/4] + VALUE[(N/4) + 1])/ 2, i.e., the mean of the number in the list at positions N/4 and (N/4)+1. If N/4 is not an integer, then the lower quartile value is VALUE[max(N/4)], i.e., the number in the list at the position (smallest integer greater than N/4). Although these formulas seem harmlessly simple, the comparisons for integer values were quite extensive in the final code because the following code did NOT work:

```
LOWER_INTEGER  : float; LOWER_POSITION : float;
LOWER_POSITION := N/4.0;
LOWER_INTEGER :=  integer( LOWER_POSITION );
if ( LOWER_POSITION = LOWER_INTEGER ) then
...
```

An obvious failure point is the type conversion in the second executed line. The programmer is promised "The conversion of a real value to an integer type rounds to the nearest integer;" but "if the operand is half-way between 2 integers, rounding may be up or down." (Ada L.R.M., 4-22) Even a simple fix by inserting "integer (LOWER_INTEGER + 0.5)" in the second line will not work necessarily on all Ada implementations because the fix depends on rounding down if the number is half-way between the two integers.

After much programming to achieve a tested, portable algorithm to find the "smallest integer greater than N/4," another pitfall can be seen in the comparison "(LOWER_POSITION = ..." Equality in real numbers is subjective and must by necessity be analyzed with someintelligence. The values 4.9999999 . . . and 5.0 are equal mathematically, but the values 0.666666 . . . and 0.67 are not. In Ada implementations floating point representations of integer numbers are sometimes of the first type (5 = 4.999999 . . .) and are sometimes more precise in lay terms (5 = 5.0000). In the first instance an algorithm will fail if it relies on a comparison such as "if (5.000 = float(5))" since the second term is seen as 4.999999 . . . . In actuality, the definition of Ada explicitly provides that the values 5.0, 3.0, and 15.0 will always be model numbers, so using the example of 5.0 will function properly. It is all values other than these three that should cause concern foı the programmer.

Ada provides a solution for these problems in its strong emphasis on packages, procedures, and overloading. Effort has gone into developing portable, generic routines for accurate floating point arithmetic, and although it is almost impossible to correct all floating point deficiencies, there are packages of floating point operations available which may be used to correct all the problems mentioned above. One such package, the GENERIC_SCIENTIFIC_COMPUTATION package developed by the DIAMOND Esprit project in Europe described in (Wallis). In this package are comprehensive floating point facilities, including ROUND_DOWN, ROUND, and ROUND_UP, which the programmer may use to specify the type of rounding to be done during a type conversion; also included are DIVIDE, which may be used for precise division of floating point numbers, and EQUAL, which may be used to test precisely the equality of floating point values (See Wallis, pp. 99-120 for a thorough description of this package). This package can be instantiated for any desired precision and claims to give precise results within the specified precision, thus supplementing Ada's power to give a wider domain in mathematical, statistical, and scientific applications.

In spite of the availability of packages such as this one, many Ada programmers may not have access to precise floating point operations. This is a dangerous situation, especially in the case of real-time programming which requires continual precision in calculations. The Ada community must be aware of the pitfalls in floating point programming and the solutions available. If accurate standard floating point packages are not provided to developers and programmers, they may be forced to rely on "home-made" mathematical functions which reduce security and reliability, or worse to rely on Ada's built-in facilities.

REFERENCES

Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, United States Department of Defense (1983).

Wallis, Peter J. L. Improving Floating-point Programming. Chichester: John Wiley and Sons (1990).

Mr. Seth M. Lowrey is an undergraduate in Computer Science at the University of Mississippi. He will graduate in May with highest honors. He will begin work with a firm in Dallas as soon as he graduates. He can be reached at the University of Mississippi, Department of Computer and Information Science, Weir Hall Rm. 302, University, MS 38677
seth@cs.OLEMISS.edu

AN ARCHITECTURE FOR AUTOMATIC COBOL-TO-ADA TRANSLATION

Ronald B. Finkbine

Computer Science Department, New Mexico Tech
Socorro, NM 87801, (505) 835-5126
e-mail: finkbine@minos.nmt.edu

ABSTRACT

Automatic conversion of various dialects of the COBOL language to ADA will allow the COBOL user community to increase usage of ADA and decrease dependence upon COBOL. The use of automatic conversion tools is the only economical method to convert applications without expensive redesign and/or reprogramming. Conversion will allow these systems to take advantage of advances in computer technology and reduce the long term costs of maintaining and enhancing COBOL software systems. This paper will discuss COBOL-to-ADA translation in terms of; translation as a method of cost-effective development of the ADA userbase, cost benefits of automatic translation, functional translation methodologies, automatic translation methodologies, concluding with a recommended architecture for COBOL-to-ADA translation.

INTRODUCTION

Today's increasing reliance on computer systems for information handling has many organizations facing the dilemma of how to modernize antiquated data processing systems. This paper discusses the advantages of conversion of existing systems, the porting of existing functionality, as an alternative to the development of new systems. Due to the large amount of COBOL code that exists in government and industry computing environments, this paper concentrates on COBOL-to-ADA conversion. It is not the intent of this paper to recommend replacement of COBOL entirely, but to investigate the issue of, cost effectively and automatically, converting existing COBOL applications to ADA with minimal programmer intervention. As the difficulty of maintaining old software systems rises and the cost of writing a new line of code passes $200[1], automated system conversion is becoming an attractive option for system migration.

DEVELOP ADA USERBASE

Government officials responsible for ADA usage are attempting to increase the number of projects and people using ADA for all forms of programming; MIS, real time, and scientific applications. This section will discuss the various opportunities becoming available to increase the use of ADA, including programmer distribution and the increasing cost of COBOL.

The intent of the DoD in developing ADA was for all new software systems to be completed in, and maintenance programming requiring substantial changes be converted to ADA unless adequate justification is provided. Despite this requirement, few projects are being implemented in ADA, producing few experienced ADA programmers and incurring ever higher costs in software system maintenance (high COBOL-induced costs due to the nature of the language and high ADA-induced costs due to few available programmers). Currently, experienced ADA programmers are more expensive than experienced COBOL programmers and will remain so until increased demand for ADA programmers force an increased supply and corresponding costs drop.

Maintenance costs of existing systems will continue to escalate as current COBOL programmers age and retire, fewer experienced COBOL programmers are being produced by industry and academia, the total number of programmers being produced by academia is falling, and recent graduates are less apt to desire to program in COBOL when jobs using other languages are available. In addition to increasing maintenance costs, the amount of program maintenance for COBOL applications will dramatically increase this decade due to the nature of COBOL itself.

On January 1, 2000, approximately 75 %[2] of all COBOL programs are going to produce incorrect results due to COBOL compilers returning the current date in YYMMDD format. The severity of this problem has not reached system users. If the majority of programs in a software system need to be edited for inventory expiration or interest computations, and management has indicated interest in converting to ADA, why not just convert the system?

The time to concentrate on COBOL-to-ADA transition is in the decade of the 1990's. This is an unprecedented opportunity for expansion of the ADA userbase. With increasing costs of maintenance, growing programmer dissatisfaction with COBOL, and the coming date forced edits, the pressure to migrate implementation languages will increase.

COST BENEFITS

When viewing methods of porting system functionality, cost is usually the deciding factor. However, cost estimates for software development are generally a risky endeavor and include a large margin of error, dependent upon the size of the project and the maturity of the technology utilized. System conversion is a relatively undeveloped field of computer science, and few studies, text, or published papers are available for research. However, one series of studies performed in the late 1970's were referenced in this paper. Various authors have developed equations that approximate the number of effort (in man-months) required for the average software development project.

$$\text{redesign and reprogram } E = 5.2 * (L\text{^}0.91) \qquad (1)^{3}$$

$$\text{reprogram } E = 2.6 * (L\text{^}0.91) \qquad (2)^{4}$$

$$\text{conversion } E = 7.14 * (L\text{^}0.47) \qquad (3)^{5}$$

where L is the number of lines of code in thousands.

The equations for redesign and reprogramming were developed over a sample of 60 completed projects and the conversion equation was developed over a sample of 31 projects[6]. The conversion equation was developed by observing manual conversion projects, no automated conversion was attempted. The equations illustrate the savings in overall system cost possible when utilizing system conversion rather than system development.

The analysis of data from converted, reprogrammed, and redesigned systems demonstrates the costs that can be saved by reuse of specifications and/or source code. For example, consider a system of 100,000 lines of code (L = 100). The expected effort (in man-months) for redesign and reprogramming would be 343.6. for reprogramming from past specifications would be 171.88, and for conversion from existing software programs would be 62.2. This estimate is for manual software conversion, so automated conversion should offer even more savings. In general, the fastest, safest, and least expensive method of porting systems is conversion, especially if the conversion includes the use of automated tools.

FUNCTIONAL TRANSLATION METHODOLOGIES

There are three methods for porting system functionality; redesign and reprogramming, reprogramming from existing system specifications, and software conversion.

The redesign of an existing system is equivalent to discarding the entire existing system and developing the system again, including functional description (what the user expects), system description (design of the system including program name, program function, and description of all files used), program description (detailed description of each program and file), and actual programming (target language code). This is very expensive in terms of programming time and user time if the users need to be interviewed and all information processes reviewed to determine requirements.

The reprogramming of existing systems involves writing new target language source code from an existing functional description or specification determined from previous user interviews and information process reviews. The cost for this type of activity is considerably less than a complete redesign and reprogramming since the functional description is reused.

The conversion of existing code is, simply, the altering of language statements from the source language to the target language. For example, most languages have statements for printing data for the user. Some languages use the word "PRINT" to signify an output statement, but words such as "WRITE" and "OUTPUT" serve the same purpose in other languages. Substituting words of similar statements among different languages is the easy part of system conversion. More difficult is determining the intent of the original programmer when that one statement was written.

A program is more than a list of statements, one can translate every statement in a program and still not fully translate the total functionality of the program. Some languages have features in support of a statement that cannot be translated in only one statement. For example, COBOL data-type conversion (from alphabetic letters to computational numbers, or vice versa) is automatic, the programmer simply moves the data between variables and the conversion is complete. However, other languages such as PASCAL and ADA are very rigid in their type checking and data type conversion involves a number of function calls (special functions made available within the language). This type of interpretation is not straight forward and dependent upon the hidden features of the implementation and target languages.

It is apparent that code translation would require a small amount of user time (a small amount of final testing would be necessary), but the effective use of programmer time is essential also. To better utilize programmer time and maintain consistency of translation across a software system using many programmers, automated tools should be used.

AUTOMATIC TRANSLATION METHODOLOGIES

In developing a software translation system, three system architectures will be considered; one translator for each COBOL dialect and implementation machine, a translator-generator, and an expert-system-based translator for all COBOL dialects.

The approach of one translator for each dialect and implementation machine clearly would be difficult to control and maintain due to the large number of implementation machines and COBOL dialects. However, building translators for certain well chosen dialects and machines is possible. For example, translators could be build for the standard COBOL implementations on popular systems; IBM mainframes, DEC and HP minicomputers, and CDC mainframes. This small set of set of translators would allow migration paths for a large portion of the existing COBOL code. This would not allow for migration of other machines (PRIME, IBM PC's, etc.) or for nonpopular COBOL vendors on popular systems.

The translator-generator would be similar to compiler-generators in use today. It would accept an input file of tokens and production rules, and using the rules, translate the source language tokens into target language tokens. The intent of software generators is to simplify the coding process, however, with the many options available on COBOL executable statements, the coding of the production rules would be very complicated since the programmer would have to code production rules for the source language, the target language, and the mapping from source to target.

The expert-system-based translator would be a complex software system and under constant revision since new versions of COBOL continue to be released by vendors. This system would process entire COBOL software systems, analyzing a set of programs as a group, not individually. This would allow the conversion system to do a number of specific system-wide functions: enforce variable name consistency, determine data file usage, and define database schemas.

This system would need to be interactive with the programmer, but record previous decisions to maintain consistency of translation. Decisions would need to be made by the system, especially when differing dialects of COBOL react completely differently when presented with the same program (i.e. the sort order problem between EBCDIC and ASCII machines). An additional problem is vendor applications that use preprocessors to implement their systems. For example, CICS and various relational databases have embedded commands in COBOL source files that are translated into pure COBOL by their preprocessors. Some mechanism would be needed to pass through these commands and hope the vendor has an ADA preprocessor.

The best type of architecture for an expert-system-based translator might be a blackboard architecture with knowledge sources for each statement (MOVE, ADD, etc.). Compiler technology would be included, but this type of system is too large and complex for extensive use of current compiler-generators. The translation system would translate individual COBOL programs, as well as implement system-wide functions. Records should be kept of all decisions by programmers using the system, notifying users when inconsistencies in translation are introduced.

FUTURE DIRECTIONS

A computer scientist should view algorithms and functions as constant and their language expression as transitory. For example, the factorial function (algorithm) is constant, but its language expression is variable. The FORTRAN, PASCAL, and COBOL code that calculates a factorial is not word for word equivalent, though each implementation might produce correct and equivalent results. The user does not care in what language a program is written, just in the results.

ADA has been developed for, and by, the DoD and, for the foreseeable future, will remain the DoD standard. But what of the future? Will Visual ADA be developed? Relational ADA? What language migration will be taking place 20 years from now and how will it be performed?

In the future, as the pressure to convert to modern languages increases, the migration among languages will increase, spurring development of automated tools for translation and data-conversion. "smart" editors, and various other tools to assist the programmer.

BIBLIOGRAPHY

[1] Poos, Bob, "ARMY'S MANAGEMENT FACES CRITICAL SHORTAGE OF SOFTWARE EXPERTISE", Federal Computer Week, May, 28, 1990.

[2] Xenakis, John, "PREPARING FOR 2000", Informationweek, 26 February, 1990.

[3] Brandon, D. H., "COMMERCIAL SOFTWARE", in "SOFTWARE PORTABILITY", Cambridge University Press, Chapter VI, p. 1977.

[4] Brown, P. J., ed., "SOFTWARE PORTABILITY", Cambridge University Press, 1977.

[5] Schneider, V., "PREDICTION OF SOFTWARE EFFORT AND PROJECT DURATION - FOUR NEW FORMULAS", SIGPLAN Notices. Vol. 13, No. 6, 1978, pp 45-59.

[6] Wolberg. John, "CONVERSION OF COMPUTER SOFTWARE", Prentice-Hall, 1981, p. 29.

The author received the B.S. and M.S. degrees in Computer Science from Wright State University, Dayton, Ohio, in 1985 and 1990, respectively. He is currently a Phd student at New Mexico Tech, and can be reached at the Computer Science Dept., New Mexico Tech, Socorro, NM 87801, (505) 835-5126, and e-mail at finkbine@minos.nmt.edu.